

Notes on Modern C++

(standard c++11, 14, 17, 20)

Gyubeom Edward Im*

July 28, 2024

Contents

1	Introduction	3
2	Intermediate	13
2.1	Temporary	13
2.1.1	return-by-value vs return-by-reference	13
2.1.2	temporary and casting	14
2.2	Conversion	14
2.2.1	Conversion constructor, conversion operator	15
2.2.2	explicit constructor	16
2.2.3	explicit conversion operator	17
2.2.4	explicit(bool), c++20	18
2.2.5	Conversion example: nullptr	19
2.2.6	Conversion example: return type resolver	19
2.2.7	Lambda expression and conversion	19
2.3	Constructor	20
2.3.1	Base from member idioms	21
2.3.2	Constructor and virtual function	22
2.4	This call	22
2.5	Member function pointer	24
2.5.1	std::invoke, std::mem_fn	25
2.6	Member data pointer	25
2.7	Implement custom max	26
2.7.1	Add compare operator	28
2.7.2	c++20 ranges algorithm	29
2.8	Size of member function pointer	29
2.9	new, delete and placement new	31
2.9.1	Using placement new	32
2.9.2	vector and placement new	32
2.10	Trivial constructor	33
2.10.1	Trivial default constructor	33
2.10.2	Trivial copy constructor	33
2.11	Type deduction	34
2.11.1	Auto type deduction	35
2.11.2	Array Name	36
2.12	Rvalue & forwarding & reference	37
2.12.1	Lvalue vs Rvalue	37
2.12.2	Reference & Overloading	38
2.12.3	Reference collapsing	39
2.12.4	Forwarding reference	40
2.13	Move semantics	41
2.13.1	Move constructor	41
2.13.2	std::move	42

*blog: alida.tistory.com, email: criterion.im@gmail.com

2.13.3	Move and noexcept	42
2.13.4	Default move constructor	44
2.13.5	Rule of 3/5/0	44
2.14	Perfect forwarding	45
2.14.1	Using forwarding refernce	46
2.14.2	Variadic parameter template	46
3	References	47
4	Revision log	47

1 Introduction

본 포스트는 필자가 Modern C++을 공부하면서 중요하다고 생각되는 내용들을 정리한 포스트이다. 각 c++ 표준별로 개선된 사항에 대해 정리하면 다음과 같다. 대부분의 내용은 [1],[2]를 참고하여 작성하였다.

- **c++11:** 해당 버전 이전을 Legacy c++라고 하고, 이후를 Modern c++이라고 구분할 만큼 많은 변화가 있었다. rvalue reference(&&)와 이동 생성자, 이동 대입 연산자, auto, lambda, constexpr 등 많은 개념이 도입되었다.

- 그외에도 nullptr, char16_t, char32_t 타입 추가,
 - 중괄호 초기화,
 - 멤버 선언부 초기화,
 - range-based for문,
 - default, delete, override, final 키워드 추가, 생성자 위임, 생성자 상속,
 - 명시적 형변환,
 - noexcept, decltype 추가,
 - perfect forwarding,
 - static _assert(),
 - 가변 템플릿,
 - 런타임 성능 개선과 컴파일 타임 프로그래밍, 코딩 컨벤션 강화 등 많은 부분이 추가되었다.

- **c++14:** c++11에 추가된 내용을 보강하였다. 리턴 타입 추론으로 c++11의 후행 리턴 개념을 보강하였고 constexpr 함수 제약을 완화하여 컴파일 타임 함수 작성 편의성을 향상하였다.

- 그외에도 variable template, decltype(auto), ‘[[deprecated]]’ 키워드,
 - 람다 캡쳐 초기화, 일반화된 람다 표현식 등의 내용이 추가되었다.

- **c++17:** 기존 컴파일러에 의존하던 코드 최적화를 임시 구체화와 복사 생략 보증으로 표준화하였고 if constexpr과 클래스 템플릿 인수 추론으로 컴파일 타임 프로그래밍을 강화하였다.

- 그외에도 inline variable, auto의 중괄호 초기화 특수 추론 규칙 개선,
 - enum의 중괄호 직접 초기화 허용, 람다 캡쳐시 *this 이용,
 - constexpr 람다 표현식,
 - static _assert()의 메세지 생략,
 - noexcept 함수 유형 포함,
 - Fold 표현식,
 - 비타입 템플릿 인자에서 auto 허용,
 - 16진수 부동 소수점 리터럴, ‘[[fallthrough]]’, ‘[[nodiscard]]’, ‘[[maybe_unused]]’ 키워드가 추가되었다.

- **c++20:** concept, requires가 추가되어 코딩 컨벤션이 강화되었으며 모듈로 컴파일 속도가 향상되었다. 코루틴으로 함수의 일시 정지가 가능해졌으며 삼중 비교 연산자가 추가되어 비교 연산자 구현이 간단해졌다. 축약된 함수 템플릿으로 auto를 함수의 인자로 사용할 수 있고, 람다 표현식에서 템플릿 인자 자원으로 람다 표현식의 일반화 프로그래밍이 강화되었다. 그리고 range-based for문에서 초기식을 사용할 수 있어 반복문 작성이 좀 더 쉬워졌다.

- 그외에도 컴파일 타임 프로그래밍 (consteval 함수, constinit, constexpr 함수)의 추가 제약이 완화되었으며
 - 템플릿 인수 추론 시 initializer_list가 개선되었고
 - 람다 표현식에서 파라미터 팩 지원, 캡쳐에서 구조화된 바인딩 지원, 상태없는 람다 표현식의 기본 생성과 복사 대입 지원, 미평가 표현식에서도 람다 표현식 허용,
 - explicit(bool) 추가,
 - 인라인 네임스페이스와 단순한 중첩 네임스페이스 결합, 지명 초기화,
 - new[]에서 중괄호 집합 초기화로 배열 크기 추론,
 - using enum 추가, __VA_OPT__, __has_cpp_attribute() 매크로 함수 추가, ‘[[nodiscard]]’의 생성자 지원, ‘[[nodiscard("reason")]]’, ‘[[likely]]’, ‘[[unlikely]]’, ‘[[no_unique_address]]’가 추가되었다.

Runtime programming	
move (c++11)	(c++11) <ul style="list-style-type: none"> 우측값 참조(&&)와 이동 생성자, 이동 대입 연산자가 추가되어 이동 연산을 지원하며, 임시 객체 대입 시 속도가 향상되었다. 전달 참조가 추가되어 포워딩 함수에서도 효율적으로 함수 인자를 완벽하게 전달할 수 있다.
unrestricted unions (c++11)	(c++11) <ul style="list-style-type: none"> 무제한 공용체(Unrestricted unions)가 추가되어 공용체 멤버에서 생성자/소멸자/가상 함수 사용 제한이 풀렸으며, 메모리 절약을 위한 코딩 자유도가 높아졌다.
Temporary materialization & Copy elision (c++17)	(c++17) <ul style="list-style-type: none"> 임시 구체화(Temporary materialization)와 복사 생략 보증(Copy elision)을 통해 컴파일러의 존적이었던 생성자 호출 및 함수 인수 전달 최적화, 리턴값 최적화 등이 표준화되었다.

Compile time programming	
constexpr (c++11)	(c++11) <ul style="list-style-type: none"> constexpr이 추가되어 컴파일 타임 프로그래밍이 강화되었다. (c++14) <ul style="list-style-type: none"> constexpr 함수의 제약이 완화되어 지역 변수, 2개 이상 리턴문, if, for, while 등을 사용할 수 있게 되었다. (c++17) <ul style="list-style-type: none"> if constexpr이 추가되어 조건에 맞는 부분만 컴파일하고 그렇지 않은 부분은 컴파일에서 제외할 수 있다. (c++20) <ul style="list-style-type: none"> consteval 함수가 추가되어 컴파일 타임 함수로만 동작할 수 있다. constinit이 추가되어 전역 변수, static 전역 변수, 정적 멤버 변수를 컴파일 타임에 초기화할 수 있다. constexpr 함수 제약 완화가 보강되어 가상 함수, dynamic_cast, typeid(), 초기화되지 않은 지역 변수, try-catch(), 공용체 멤버 변수 활성 전환, asm 등을 사용할 수 있다.
static_assert (c++11)	(c++11) <ul style="list-style-type: none"> static_assert가 추가되어 컴파일 타임 진단이 가능해졌다. (c++17) <ul style="list-style-type: none"> static_assert() 메시지의 생략을 지원한다.

advanced template (c++11)	<p>(c++11)</p> <ul style="list-style-type: none"> 가변 템플릿 파라미터 팩이 추가되어 가변 인자(...)와 같이 갯수와 타입이 정해지지 않은 템플릿 인자를 사용할 수 있다. sizeof...() 연산자가 추가되어 가변 템플릿에서 파라미터 팩의 인자수를 구할 수 있다. extern으로 템플릿을 선언할 수 있으며 템플릿 인스턴스 중복 생성을 없앨 수 있다. 템플릿 오른쪽 꺽쇠 괄호 파싱을 개선하여 템플릿 인스턴스화 시 >가 중첩되어 »와 같이 되더라도 공백을 추가할 필요가 없어졌다. <p>(c++14)</p> <ul style="list-style-type: none"> variable template이 추가되어 변수도 템플릿으로 만들 수 있다. <p>(c++17)</p> <ul style="list-style-type: none"> 클래스 템플릿 인수 추론이 추가되어 함수 템플릿처럼 템플릿 인스턴스화시 타입을 생략할 수 있다. 클래스 템플릿 인수 추론 사용자 정의 가이드가 추가되어 클래스 템플릿 인수 추론 시 컴파일러에게 가이드를 줄 수 있다. 비타입 템플릿 인자에서 auto를 허용한다. Fold 표현식이 추가되어 가변 템플릿에서 파라미터 팩을 재귀적으로 반복하여 전개할 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> 축약된 함수 템플릿이 추가되어 auto 타입을 사용할 수 있다. 사실 상 함수 템플릿의 간략한 표현이다. 비타입 템플릿 인자 규칙이 완화되어 실수 타입과 리터럴 타입을 사용할 수 있다. 클래스 템플릿 인수 추론 시 initializer_list인 경우가 개선되어 std::vector v1,2,3 처럼 템플릿 인자를 명시하지 않아도 된다.
concept (c++20)	<p>(c++20)</p> <ul style="list-style-type: none"> concept과 requires가 추가되어 템플릿 인자나 auto에 제약 조건을 줄 수 있다.

Coding convention enhancement

advanced type and literals ([c++11](#))

([c++11](#))

- 타입 카테고리를 수립하여 컴파일 타임 프로그래밍이나 템플릿 메타 프로그램이 시 코딩 컨벤션을 강화할 수 있다.
- using을 이용한 타입 별칭이 추가되어 typedef보다 좀 더 직관적인 표현이 가능해졌다.
- nullptr 리터럴이 추가되어 포인터에 안전한 코딩 컨벤션이 가능해졌다.
- long long 타입이 추가되어 최소 8byte 크기를 보장한다.
- ll, ull, LL, ULL 리터럴이 추가되어 long long용 정수형 상수를 제공한다.
- char16_t, char32_t 타입이 추가되어 UTF-16 인코딩, UTF-32 인코딩을 모두 지원한다.
- u8 "", u", U", u"(char), U"(char) 리터럴이 추가되어 유니코드를 지원하는 char16_t, char32_t 타입용 문자 상수를 제공한다.
- R"()" 리터럴이 추가되어 개행이나 이스케이프 문자를 좀 더 편하게 입력할 수 있다.

([c++14](#))

- 이진 리터럴이 추가되어 0b, 0B 접두어로 이진수 상수를 표현할 수 있다.
- 숫자 구분자가 추가되어 1'000'000과 같이 작은 따옴표 '를 숫자 사이에 선택적으로 넣을 수 있어 가독성이 좋아졌다.

([c++17](#))

- 16진수 부동 소수점 리터럴이 추가되어 0xA, 9p11과 같이 16진수로 실수를 표현할 수 있다.
- u8"(char) 리터럴이 추가되어 유니코드를 지원하는 1byte 크기의 문자 상수를 지원한다.

([c++20](#))

- char8_t 타입이 추가되어 UTF-8 인코딩 문자를 지원한다.
- 정수에서 2의 보수 범위를 보장한다.
- 사용자 정의 리터럴 인자 규칙에 char8_t가 추가되었다.

noexcept ([c++11](#))

([c++11](#))

- noexcept가 추가되어 함수의 예외 방출 여부를 보증하며 소멸자는 기본적으로 noexcept로 작동한다.
- noexcept 연산자가 추가되어 해당 함수가 noexcept인지 컴파일 타임에 검사할 수 있다.

([c++17](#))

- noexcept가 함수 유형에 포함되어 예외 처리에 대한 코딩 컨벤션을 좀 더 단단하게 할 수 있다.

explicit type conversion (c++11)	<p>(c++11)</p> <ul style="list-style-type: none"> explicit 형변환 연산자가 추가되어 명시적으로 형변환할 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> explicit(bool)이 추가되어 특정 조건일 때만 explicit으로 동작하게 할 수 있다.
attribute (c++11)	<p>(c++11)</p> <ul style="list-style-type: none"> attribute가 추가되어 컴파일러에게 부가 정보를 전달하는 방식을 표준화하였다. <p>(c++14)</p> <ul style="list-style-type: none"> '[[deprecated]]'가 추가되어 소멸 예정인 것을 컴파일 경고로 알려준다. <p>(c++17)</p> <ul style="list-style-type: none"> '[[fallthrough]]'가 추가되어 switch()에서 의도적으로 break를 생략하여 다음 case로 제어를 이동시킬 때 발생하는 컴파일 경고를 차단할 수 있다. '[[nodiscard]]'가 추가되어 리턴값을 무시하지 않도록 컴파일 경고를 해준다. '[[maybe_unused]]'가 추가되어 사용되지 않은 객체의 컴파일 경고를 차단할 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> '[[nodiscard]]'의 생성자 지원, '[[nodiscard("reason")]]'이 추가되었다. '[[likely]]', '[[unlikely]]'가 추가되어 컴파일러에게 최적화 힌트를 줄 수 있다. '[[no_unique_address]]'가 추가되어 아무 멤버 변수가 없는 객체의 크기를 최적화할 수 있다.

Coding convenience enhancement	
advanced namespace (c++11)	<p>(c++11)</p> <ul style="list-style-type: none"> 인라인 네임스페이스가 추가되어 API 버전 구성이 편리해졌다. <p>(c++14)</p> <ul style="list-style-type: none"> 단순한 중첩 네임스페이스가 추가되어 ::로 표현할 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> 인라인 네임스페이스와 단순한 중첩 네임스페이스를 결합하여 표현할 수 있다.

advanced initialization ([c++11](#))

([c++11](#))

- 중괄호 초기화를 제공하여 클래스, 배열, 구조체 구분없이 중괄호로 일관성있게 초기화를 할 수 있으며 초기화 패싱 오류를 해결하였다.
- 중괄호 복사 초기화로 함수 인수 전달, 리턴문 작성을 간결화할 수 있다.
- 중괄호 초기화시 인자의 암시적 형변환을 일부 차단하여 코딩 컨벤션이 개선되었다.
- initializer_list가 추가되어 vector 등 컨테이너의 초기 요소 추가가 간편해졌다.
- 멤버 선언부 초기화가 추가되어 non-static 멤버 변수의 초기화가 쉬어졌다.

([c++14](#))

- non-static 멤버 선언부 초기화 시 집합 초기화를 허용한다.

([c++20](#))

- 지명 초기화가 중괄호 집합 초기화 시 변수명을 지명하여 값을 초기화할 수 있다.
- 비트 필드 선언부 초기화가 추가되었다.
- new[]에서 중괄호 집합 초기화로 배열 크기 추론이 추가되어 배열 크기를 명시하지 않아도 된다.

advanced control statement ([c++11](#))

([c++11](#))

- range-based for()가 추가되어 컨테이너 요소의 탐색 처리가 쉬워졌다.

([c++17](#))

- 초기식을 포함하는 if(), switch()가 추가되어 함수 리턴값을 평가하고 소멸하는 코드가 단순해졌다.

([c++20](#))

- range-based for()에서 초기식이 추가되었다.

advanced class (c++11)	(c++11) <ul style="list-style-type: none"> default, delete가 추가되어 암시적으로 생성되는 멤버 함수의 사용 여부를 좀 더 명시적으로 정의할 수 있다. override가 추가되어 가상 함수 오버라이딩 코딩 규약이 좀 더 단단해졌다. final이 추가되어 가상 함수를 더 이상 오버라이딩 못하게 할 수 있고 강제적으로 상속을 제한할 수 있다. 생성자 위임이 추가되어 생성자의 초기화 리스트 코드가 좀 더 간결해졌다. 생성자 상속이 추가되어 부모 객체의 생성자도 상속받아 사용할 수 있어 자식 객체의 생성자 재정의 코드가 좀 더 간결해졌다. 멤버 함수 참조 지정자가 추가되어 멤버 함수에 &, &&로 lvalue로 호출될 때와 rvalue로 호출 될 때를 구분하여 오버로딩을 할 수 있다.
auto decltype trailing return type (c++11)	(c++11) <ul style="list-style-type: none"> auto와 decltype()이 추가되어 값으로부터 타입을 추론하여 코딩이 간편해졌다. 후행 리턴(trailing return)이 추가되어 함수 인자에 의존하여 리턴 타입을 결정하며 좀 더 동적인 함수 설계가 가능해졌다. <p>(c++14)</p> <ul style="list-style-type: none"> decltype(auto)가 추가되어 decltype()의 () 내 표현식이 복잡할 경우 좀 더 간결하게 작성할 수 있다. 리턴 타입 추론이 추가되어 후행 리턴 대신 auto나 decltype(auto)를 사용할 수 있다. <p>(c++17)</p> <ul style="list-style-type: none"> auto의 중괄호 초기화 특수 추론 규칙이 개선되어 auto a{1}; 시 initializer_list가 아니라 int로 추론된다.
scoped enum (c++11)	(c++11) <ul style="list-style-type: none"> 범위 있는 열거형(scoped enum)이 추가되어 이를 충돌 회피가 쉬워졌고 암시적 형변환을 차단하며 전방 선언도 지원한다. <p>(c++17)</p> <ul style="list-style-type: none"> 열거형의 중괄호 직접 초기화를 허용하여 암시적 형변환을 차단하는 사용자 정의 열거형의 사용이 좀 더 쉬워졌다. <p>(c++20)</p> <ul style="list-style-type: none"> using enum이 추가되어 열거형의 이름 없이 열거자를 유효 범위 내에서 사용할 수 있다.

lambda expression, closure (c++11)	<p>(c++11)</p> <ul style="list-style-type: none"> 람다 표현식이 추가되어 1회용 익명 함수를 만들 수 있다. <p>(c++14)</p> <ul style="list-style-type: none"> 람다 캡쳐 초기화가 추가되어 람다 표현식 내에서 사용하는 임의 변수를 정의하여 사용할 수 있다. 일반화된 람다 표현식이 추가되어 auto를 받아 마치 함수 템플릿처럼 사용할 수 있다. <p>(c++17)</p> <ul style="list-style-type: none"> 람다 캡쳐 시 *this가 추가되어 객체 자체를 복제하여 사용할 수 있다. constexpr 람다 표현식이 추가되어 람다 표현식도 컴파일 타임 함수로 만들 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> 람다 표현식에서 템플릿 인자를 지원한다. 람다 캡쳐에서 파라미터 팩을 지원한다. 람다 캡쳐에서 구조화된 바인딩을 지원한다. 상태없는 람다 표현식의 기본 생성과 복사 대입을 지원한다. 미평가 표현식에서도 람다 표현식을 허용하기 때문에 decltype() 안에서 사용할 수 있다.
advanced variables (c++17)	<p>(c++17)</p> <ul style="list-style-type: none"> 인라인 변수가 추가되어 헤더 파일에 정의된 변수를 여러 개의 cpp에서 include 하더라도 중복 정의없이 사용할 수 있다. 또한 클래스 정적 멤버 변수를 선언부에서 초기화할 수 있다.
structured bindings (c++17)	<p>(c++17)</p> <ul style="list-style-type: none"> 구조화된 바인딩(structured bindings)이 추가되어 배열, pair, tuple, class 등 내부 요소나 멤버 변수에 쉽게 접근할 수 있다.
advanced operator (c++20)	<p>(c++20)</p> <ul style="list-style-type: none"> 삼중 비교 연산자가 추가되어 비교 연산자 구현이 간소화 되었다. 삼중 비교 연산자를 default로 정의할 수 있다. 비트 쉬프트 연산자의 기본 비트가 표준화되어 « 1은 곱하기 2의 효과가 있는 비트(즉, 0)으로 채워지고 » 1은 나누기 2의 효과가 있는 비트 (즉, 양수면 0, 음수면 1)로 채워진다.

module (c++20)	(c++20) <ul style="list-style-type: none"> 모듈이 추가되어 전처리 사용 방식을 개선하여 컴파일 속도를 향상시키고, include 순서에 따른 종속성 문제, 선언과 정의 분리 구성의 불편함, 기호 충돌 문제를 해결하였다.
coroutine (c++20)	(c++20) <ul style="list-style-type: none"> 코루틴이 추가되어 함수의 일시 정지 후 재개가 가능해졌다.
misc	(c++11) <ul style="list-style-type: none"> alignas(), alignof()가 추가되어 메모리 정렬 방식을 표준화하였다. 가변 매크로가 추가되어 c언어와 호환성이 높아졌다. 멤버의 sizeof() 시 동작이 개선되어 객체를 인스턴스화하지 않아도 객체 멤버의 크기를 구할 수 있다. <p>(c++17)</p> <ul style="list-style-type: none"> __has_include가 추가되어 include 하기 전에 파일이 존재하는지 확인할 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> __VA_OPT__ 가 추가되어 가변 인자가 있을 경우에는 팔호 안의 값으로 치환하고 없을 경우에는 그냥 비워둔다. __has_cpp_attribute() 매크로 함수가 추가되어 c++11부터 추가된 attribute가 지원되는지 확인할 수 있다. 언어 지원 테스트 매크로가 추가되어 c++11부터 추가된 언어 기능을 지원하는지 테스트할 수 있다.

deprecated & removed

(c++11)

- 동적 예외 사양은 deprecated되었다. 예외를 나열하는 것 보다 noexcept로 예외를 방출하느냐 안하느냐만 관심을 둔다.
- export 템플릿은 제대로 구현한 컴파일러는 드물고 세부 사항에 대한 의견이 일치하지 않아 c++11부터 완전히 remove 되었다.

(c++17)

- 동적 예외 사양 관련해서 throw()가 deprecated되었다. 이제 noexcept만 사용해야 한다.
- &&, &=등이 특수기호가 없는 인코딩을 사용하는 곳을 위해 제공했던 trigraph가 remove되었다.
- 변수를 CPU 레지스터에 배치하도록 힌트를 주는 register 가 deprecated되었다.
- bool의 증감 연산이 deprecated되었다.

(c++20)

- 람다 캡쳐에서 [=] 사용 시 this의 암시적 캡쳐가 deprecated되었으므로 명시적으로 작성해야 한다.
- volatile 일부가 deprecated되었다.

2 Intermediate

2.1 Temporary

```
1 class Point{
2     int x,y;
3     public:
4
5     Point(int x, int y) : x(x), y(y) { std::cout << "Point(int,int)" << std::endl; }
6     ~Point() { std::cout<<"~Point()" << std::endl; }
7 };
8
9 Point pt(1,2); // --> 일반 객체
10 Point (1,2); // --> 임시 객체
11
12 pt.x = 10;           // ok
13 Point(1,2).x = 10; // error (rvalue라고 부름)
14 Point(1,2).set(10,20) // ok (멤버함수는 불러와지므로 상수는 아님)
15
16 Point *p1 = &pt;      // ok
17 Point *p2 = &Point(1,2) // error (임시객체는 포인터로 가르킬 수 없다)
18
19 Point& r1 = pt;      // ok
20 Point& r2 = Point(1,2); // error(임시객체는 참조로 가르킬 수 없다)
21 Point&& r3 = Point(1,2); // ok (rvalue reference 문법)
22
23 const Point &r4 = Point(1,2) // ok(상수 참조는 가능, 일반 객체로 승격되는 효과)
```

객체를 함수 인자로만 사용한다면 임시객체로 전달하는게 효율적일 수 있다. (const reference로 받아야만 함)

```
1 void foo(const Point& pt) { std::cout << "foo" << std::endl; }
2
3 int main() {
4     Point pt(1,2);
5     foo(pt);           // pt는 foo에 넣기 위해서만 생성한 객체이므로 함수가 불린 후 바로 파괴되는게 좋다.
6
7     foo(Point(1,2)) // 임시 객체를 활용하자
8     foo( {1, 2} )   // 이렇게 전달하는 것도 가능하다. 컴파일러가 Point1,2로 바꿔줌
9     std::cout<<"-----"<<std::endl;
10 }
```

```
1 void foo(const std::string& s) { }
2 void goo(std::string_view s) { } // call-by-value임에 유의!
3
4 int main() {
5     foo("Practice make perfect"); // 컴파일러에서 string("Practice make perfect")로 변환해줌
6
7     goo("Practice make perfect"); // string_view는 문자열의 복사본을 생성하지 않고 이미 상수 메모리의 존재하는
8     문자열을 가르킨다.
9 }
```

2.1.1 return-by-value vs return-by-reference

```
1 Point pt(1,2);
2 Point f3() { return pt; }
3 Point& f4() { return pt; }
4
5 Point& f5() {
6     Point pt(1,2);
7     return pt;    // error. 지역객체를 return-by-reference하면 안됨!
8 }
9
10 int main() {
```

```
11     f3().x = 10; // error. (return-by-value는 복사본이 생성되어 임시객체이므로 rvalue임)
12     f4().x = 10; // ok. pt.x = 10
13 }
```

```
1 class Counter {
2     int count{0};
3
4     Counter& increment() { //return-by-reference로 하는걸 잊으면 안됨!
5         ++count;
6         return *this;
7     }
8     int get() const { return count; }
9 }
10
11 int main() {
12     Counter c;
13     c.increment().increment().increment();
14     std::cout << c.get() << std::endl;
15 }
```

2.1.2 temporary and casting

```
1 struct Base {
2     int value = 10;
3
4     Base() = default;
5     Base(const Base&b) : value(b.value)
6     { std::cout << "copy constructor" << std::endl; }
7 }
8
9 struct Derived : public Base {
10     int value = 10;
11 }
12
13 int main() {
14     Derived d;
15
16     std::cout << d.value << std::endl; // 20
17     std::cout << static_cast<Base&>(d).value << std::endl; // 10 good. 참조 캐스팅을 통해 d를 Base로
18     생각하게 하여 값을 가져온다.
19     std::cout << static_cast<Base>(d).value << std::endl; // 10 no. Base 클래스의 복사본이 생성되며 거기서
20     값을 가져온다.
21     static_cast<Base&>(d).value = 100;
22     static_cast<Base>(d).value = 100; // error. 캐스팅은 항상 reference로 하자!
}
```

2.2 Conversion

이번 섹션에서는 객체 변환에 대한 다양한 문법과 기법에 대해 설명한다.

- 변환 연산자, 변환 생성자, explicit 생성자의 개념
- safe bool 개념과 explicit 변환 연산자에 대해 살펴본다.
- nullptr과 return type resolver 기술
- temporary proxy 기술
- lambda expression과 함수 포인터 변환

2.2.1 Conversion constructor, conversion operator

```
1 class Int32 {
2     int value;
3     public:
4     Int32() : value(0) {}
5 };
6
7 int main() {
8     int pn;           // primitive type
9     Int32 un; // user type 값을 지정해주지 않아도 쓰레기값이 아닌 0으로 초기화된다
10
11    pn = un;
12 }
```

int를 대체하기 위한 Int32 클래스에 대해 알아보자. 새로 정의한 타입은 기존에 타입과도 호환이 되는게 좋기 때문에 pn = un과 같이 서로 변환이 가능해야 한다.

pn=un이 호출된 순간 컴파일러는 un.operator int() 함수를 찾게 되는데 이러한 연산자를 **변환 연산자(conversion operator)**라고 한다.

```
1 operator TYPE() { 변환 연산자
2     return value;
3 }
```

변환 연산자는 반환 타입이 함수 이름에 포함되어 있으므로 반환 타입을 표기하지 않는다.

```
1 class Int32 {
2     int value;
3     public:
4     Int32() : value(0) {}
5     operator int() const { return value; } const를 추가하여 상수 객체 또한 동작하도록 해준다
6 };
7
8 int main() {
9     int pn;
10    Int32 un;
11    const Int32 un2;
12
13    pn = un;
14    pn = un2;
15 }
```

다음으로 un = pn과 같이 반대의 경우를 보자. 이런 경우 컴파일러는 un.operator=(int) 대입연산자가 존재하는지 먼저 검사한다. 만약 없다면 Int32(int)와 같이 default 대입 연산자가 있는지 검사한다.

```
1 class Int32 {
2     int value;
3     public:
4     Int32() : value(0) {}
5     Int32(int n) : value(n) {} 변환 생성자
6     operator int() const { return value; }
7 };
8
9 int main() {
10    int pn;           // primitive type
11    Int32 un; // user type 값을 지정해주지 않아도 쓰레기값이 아닌 0으로 초기화된다
12    const Int32 un2;
13
14    pn = un;
15    pn = un2;
16    un = pn; // 변환 생성자 호출!
17 }
```

변환 생성자에 대해 좀 더 자세히 알아보자

```

1 class Int32 {
2     int value;
3     public:
4     Int32(int n) : value(n) {}
5 };
6
7 int main() {
8     Int32 n1(3);      // 1. direct initialization
9     Int32 n2 = 3;    // 2. copy initialization
10    Int32 n3{3};     // 3. direct init. (c++11)
11    Int32 n4 = {3}; // 4. copy init. (c++11)
12
13    n1 = 3; // conversion (int -> Int32)
14 }

```

변환 생성자가 있는 경우 위와 같이 변환 이외에도 4가지 초기화 코드를 만들 수 있다. 이 중 2번에 대해 자세히 알아보자.

`Int32 n2 = 3`이 호출되면 컴파일러는 이를 `Int32 n2 = Int32(3)`으로 변환한다. 이는 임시 객체(temporary)이므로 **c++98** 시절까지는 복사 생성자를 통해 `n2`에 대입되지만 **c++11** 이후부터는 move 생성자를 통해 `n2`로 대입된다. 하지만 대부분의 컴파일러에서 최적화 옵션을 켜면 임시 객체 생성이 제거된다.

```

1 class Int32 {
2     int value;
3     public:
4     Int32(int n) : value(n) {}
5     Int32(const Int32&) = delete; // 복사 생성자 제거
6 };
7
8 int main() {
9     Int32 n1(3);
10    Int32 n2 = 3; // error! c++14까지는 에러가 발생하지만 c++17부터는 문법적으로 복사 생성을 하지 않기
11    때문에 ok
12    Int32 n3{3};
13    Int32 n4 = {3};
14
15    n1 = 3;
16 }

```

다음으로 마지막 줄의 `n1 = 3`을 살펴보자. 이는 컴파일러에 의해 `n1 = Int32(3)`으로 변환된다. 즉, 임시 객체가 생성되고 디폴트 대입 연산자를 사용해서 `n1`에 대입되는 형태이다. 대부분의 컴파일러가 최적화를 통해 임시 객체 생성이 제거되지만 **디폴트 대입 연산자는 반드시 존재해야 한다.**

```

1 class Int32 {
2     int value;
3     public:
4     Int32(int n) : value(n) {}
5     Int32(const Int32&) = delete; // 복사 생성자 제거
6     Int32& operator=(const Int32&) = delete; // 디폴트 대입 연산자 제거
7 };
8
9 int main() {
10    Int32 n1(3);
11    //Int32 n2 = 3;
12    Int32 n3{3};
13    Int32 n4 = {3};
14
15    n1 = 3; // error. 디폴트 대입 연산자가 없으면 모든 버전에 대해 에러가 발생한다.
16 }

```

2.2.2 explicit constructor

```

1 class Vector{
2     public:
3     Vector(int size) {}

```

```

4   };
5   void foo(Vector v) {}
6   int main() {
7     Vector v1(3);
8     Vector v2 = 3;
9     Vector v1{3};
10    Vector v1 = {3};

11
12    v1 = 3; // 논리적으로 벡터에 3을 넣는게 맞지 않음!
13    foo(3); // ok but no. Vector v = 3의 형태로 복사생성자가 호출됨
14 }

```

explicit 생성자를 사용하면 생성자가 암시적 변환의 용도로 사용될 수 없게 한다. 이런 경우 직접 초기화(direct initialization)만 가능하고 복사 초기화(copy initialization)은 사용할 수 없다.

```

1 class Vector{
2   public:
3     explicit Vector(int size) {}
4 };
5 void foo(Vector v) {}
6 int main() {
7   Vector v1(3);
8   Vector v2 = 3; // error
9   Vector v1{3};
10  Vector v1 = {3}; // error

11
12  v1 = 3; // error
13  foo(3); // error
14 }

```

explicit을 사용할 때는 클래스에 따라 explicit을 사용할 지 잘 판단해야 한다.

```

1 void f1(Int32 n) {}
2 void f2(Vector v) {}

3
4 f1(3); // ok. 논리적으로 맞으므로 Int32 생성자에는 explicit을 붙일 필요 없음
5 f2(3); // ok but no. 논리적으로 맞지 않으므로 Vector 생성자에는 explicit을 붙여야함

```

2.2.3 explicit conversion operator

객체의 유효성을 if문을 통해 비교하고 싶다고 하자.

```

1 class Machine {
2   int data = 10;
3   bool state = true;
4 };
5 int main() {
6   Machine m;
7   if(m) {} // 객체의 유효성을 if문을 통해 비교하고자 한다
8 }

```

컴파일을 해보면 'Machine을 bool로 변환할 수 없다'는 에러 구문이 발생하는데 이는 곧 **Machine을 bool로만 변화할 수 있으면** if문이 동작한다는 얘기와 같다.

```

1 class Machine {
2   int data = 10;
3   bool state = true;
4   public:
5     operator bool() { return state; } // bool 연산자
6 };
7
8 int main() {
9   Machine m;
10  if(m) {} // ok
11 }

```

if(m) 구문은 이제 정상적으로 작동하지만 **operator bool()**은 side effect가 많아서 사용에 주의해야 한다.
예를 들어 다음과 같은 side effect가 발생한다.

```
1 class Machine {
2     int data = 10;
3     bool state = true;
4     public:
5         operator bool() { return state; }
6     };
7
8     int main() {
9         Machine m;
10
11     bool b1 = m; // ok
12     bool b2 = static_cast<bool>(m); // ok
13     m << 10; // 문법이 이상하지만 컴파일 된다. 즉, 버그의 원인이 될 수 있다.
14     if(m) { ... }
15 }
```

이러한 문제를 해결하기 위해 c++11부터 생성자 뿐만 아니라 변환 연산자도 **explicit**을 붙일 수 있다. 이러한 기술을 **safe bool**이라고 부른다.

```
1 class Machine {
2     int data = 10;
3     bool state = true;
4     public:
5         explicit operator bool() { return state; }
6     };
7
8     int main() {
9         Machine m;
10
11     bool b1 = m; // error
12     bool b2 = static_cast<bool>(m); // ok. 명시적 변환이므로
13     m << 10; // error
14     if(m) { ... }
15 }
```

c++98부터 **explicit** 생성자 구문을 제공하였으나 c++11이 되면서 **explicit** 변환 연산자 문법이 추가되었다.
그리고 c++20에는 **explicit(bool)** 문법이 제공되었다. **explicit(bool)** 문법에 대해 알아보자.

2.2.4 **explicit(bool)**, c++20

```
1 template<class T>
2 class Number {
3     T value;
4     public:
5         explicit(true) Number(T v): value(v) {}
6     };
7
8     int main() {
9         Number n1 = 10; // error
10        Number n2 = 3.4; // error. explicit[0] true이므로 에러 발생. 만약 explicit(false)이면 컴파일이 정상적으로
11        된다
12 }
```

위 코드에서 **explicit(true)**를 하면 **expliciit** 키워드를 사용한다는 의미이고 반대로 **explicit(false)**를 하면 사용하지 않는다는 의미이다. 만약 정수형일 때만 **explicit**을 사용하지 않고 **float**일 때는 **explicit**을 사용하고 싶으면 어떻게 해야 할까?

```
1 template<class T>
2 class Number {
3     T value;
4     public:
5         explicit(!std::is_integral_v<T>)
```

```

6     Number(T v): value(v) {}
7 };
8 int main() {
9     Number n1 = 10; // ok. explicit(false)이므로
10    Number n2 = 3.4; // error. explicit(true)이므로
11 }

```

2.2.5 Conversion example: nullptr

nullptr은 포인터 초기화 시 0 대신 사용하는 C++ 키워드이다. 원래 boost 라이브러리에 있는 도구를 C++11을 만들면서 표준에 추가되었다. 이번 섹션에서는 boost에 있는 nullptr을 구현하면서 nullptr의 개념에 대해 다시 한번 숙지해보자.

```

1 void foo(int* p) {}
2 void goo(char* p) {}

3
4 struct nullptr_t {
5     template<class T>
6     constexpr operator T*() const { return 0; }
7 };
8 nullptr_t xnullptr;

9
10 int main() {
11     foo(xnullptr);
12     goo(xnullptr); // nullptr은 이미 키워드이므로 임의의 xnullptr을 정의하였다.
13 }

```

위 코드 예제와 유사하게 실제 nullptr의 타입도 std::nullptr_t이다.

2.2.6 Conversion example: return type resolver

Return type resolver는 좌변을 보고 우변의 반환 타입을 자동으로 결정하는 기술을 말한다. 다음과 같은 예제 코드를 보자.

```

1 template<class T>
2 T* Alloc(std::size_t sz) {
3     return new T[sz];
4 }

5
6 int main() {
7     int* p1 = Alloc<int>(10);
8     double* p2 = Alloc<double>(10); // <double>과 같은 꺽쇠를 사용하지 않고 바로 Alloc을 사용할 수는
9     없을까?
}

```

Alloc(10)으로 바로 받기 위해서는 Alloc이 함수가 아니라 구조체여야 한다.

```

1 struct Alloc {
2     std::size_t size;
3     Alloc(std::size_t sz) : size(sz) {}
4
5     template<class T>
6     operator T*() { return new T[size]; }
7 };
8
9 int main() {
10    int* p1 = Alloc(10);
11    double* p2 = Alloc(10); // 변환연산자에 의해 자동으로 타입이 결정되어 할당된다.
}

```

2.2.7 Lambda expression and conversion

일반적으로 람다 표현식은 auto 변수에 담아서 사용하지만 함수 포인터로도 람다 표현식을 담을 수 있다.

```

1 int main(){

```

```

2     auto f1 = [] (int a, int b) { return a + b; }
3     int (*f2)(int, int) = [] (int a, int b) { return a + b; } // 람다 표현식은 임시객체를 만들어주므로
4         임시객체.operator 함수포인터()가 호출되어 함수포인터에 할당된다.
}

```

2.3 Constructor

이번 섹션에서는 생성자의 호출 원리에 대해 살펴본다. 다음과 같은 4개의 구조체를 보자.

```

1 struct BM {
2     BM() { cout << "BM()" << endl; }
3     ~BM() { cout << "~BM()" << endl; }
4 };
5 struct DM {
6     DM() { cout << "DM()" << endl; }
7     DM(int) { cout << "DM(int)" << endl; }
8     ~DM() { cout << "~DM()" << endl; }
9 };
10 struct Base {
11     BM bm;
12     Base() { cout << "Base()" << endl; }
13     Base(int a) { cout << "Base(int)" << endl; }
14     ~Base() { cout << "~Base()" << endl; }
15 };
16 struct Derived : public Base {
17     DM dm;
18     Derived() { cout << "Derived()" << endl; }
19     Derived(int a) { cout << "Derived(int)" << endl; }
20     ~Derived() { cout << "~Derived()" << endl; }
21 };
22
23 int main() {
24     Derived d1; // call Derived::Derived()
25     Derived d2(7); // call Derived::Derived(int)
26 }

```

d1, d2을 호출하는 순간 컴파일러에 의해 자동 생성된 코드가 실행된다.

```

1 struct Base {
2     BM bm;
3     Base() : bm() { cout << "Base()" << endl; }
4     Base(int a) : bm() { cout << "Base(int)" << endl; }
5     ~Base() { cout << "~Base()" << endl; bm. BM(); }
6 };
7 struct Derived : public Base {
8     DM dm;
9     Derived() : Base(), dm() { cout << "Derived()" << endl; }
10    Derived(int a) : Base(), dm() { cout << "Derived(int)" << endl; }
11    ~Derived() { cout << "~Derived()" << endl; dm. DM(); Base(); }
12 };
13
14 int main() {
15     Derived d1; // call Derived::Derived()
16 }

```

생성자, 소멸자 호출자를 정확히 명시해주지 않아도 컴파일러가 **기반 클래스 및 멤버 데이터의 생성자(소멸자)**를 호출해주는 코드를 생성해준다.

다음으로 호출 순서를 정확히 아는 것이 중요하다. d1을 호출하면 **bm() → Base() → dm() → Derived()**가 순서대로 호출된다. 임의로 코드의 위치를 바꿔도 **사용자가 순서대로 호출 순서를 변경할 수 없다.**

또한, 컴파일러가 생성한 코드는 항상 디폴트 생성자를 호출한다. **기반 클래스나 멤버 데이터에 디폴트 생성자가 없는 경우 반드시 사용자가 디폴트가 아닌 다른 생성자를 호출하는 코드를 작성해야 한다.**

```

1 struct BM {
2     BM() { cout << "BM()" << endl; }
3     ~BM() { cout << "~BM()" << endl; }

```

```

4   };
5   struct DM {
6     //DM() { cout << "DM()" << endl; }
7     DM(int) { cout << "DM(int)" << endl; }
8     ~DM() { cout << "~DM()" << endl; }
9   };
10  struct Base {
11    BM bm;
12    //Base() { cout << "Base()" << endl; }
13    Base(int a) { cout << "Base(int)" << endl; }
14    ~Base() { cout << "~Base()" << endl; }
15  };
16  struct Derived : public Base {
17    DM dm;
18    Derived() :Base(0), dm(0) { cout << "Derived()" << endl; } // Base(0), dm(0)을 호출하지 않으면 에러
19    발생!
20    Derived(int a) : Base(0), dm(0) { cout << "Derived(int)" << endl; }
21    ~Derived() { cout << "~Derived()" << endl; }
22  };
23
24  int main() {
25    Derived d1;
26    Derived d2(7);
27  }

```

2.3.1 Base from member idioms

다음과 같이 버퍼를 사용하는 스트림 예제 코드를 보자.

```

1  class Buffer {
2    public:
3      Buffer(std::size_t sz) { cout << "initialize buffer" << endl; }
4      void use() { cout << "use buffer" << endl; }
5  };
6
7  class Stream {
8    public:
9      Stream(Buffer& buf) { buf.size(); }
10 };
11
12 int main() {
13   Buffer buf(1024);
14   Stream s(buf);
15 }

```

위 코드는 정상적으로 initialize buffer를 통해 버퍼를 초기화한 후 use buffer가 호출되어 버퍼를 사용한다.
다음으로 buffer를 멤버함수로 두고 싶은 경우를 생각해보자.

```

1  class Buffer {
2    public:
3      Buffer(std::size_t sz) { cout << "initialize buffer" << endl; }
4      void use() { cout << "use buffer" << endl; }
5  };
6  class Stream {
7    public:
8      Stream(Buffer& buf) { buf.size(); }
9  };
10
11 class StreamWithBuffer : public Stream {
12   Buffer buf(1024);
13   public:
14   StreamWithBuffer() : Stream(buf) {}
15 };
16
17 int main() {

```

```
18     StreamWithBuffer swf; // error. use buffer, initialize buffer 순으로 잘못 호출됨!
19 }
```

위 코드는 초기화되지 않은 버퍼를 사용하는 문제가 발생한다. 즉, **멤버 데이터보다 기반 클래스의 생성자가 먼저 호출되는 문제가 발생한다.**

```
1 class StreamWithBuffer : public Stream {
2     Buffer buf(1024);
3     public:
4     StreamWithBuffer() : Stream(buf), buf(1024) {} // 컴파일러에 의해 buf(1024)가 늦게 호출된다
5 }
```

이를 해결하는 방법은 다중 상속으로 해결해야 한다.

```
1 class StreamBuffer {
2     protected:
3     Buffer buf(1024);
4 };
5
6 class StreamWithBuffer : public StreamBuffer, public Stream { // 다중 상속을 통해 문제를 해결
7     public:
8     StreamWithBuffer() : StreamBuffer(), Stream(buf) {}
9 }
```

위 기술은 c++ 초기 설계 당시 ostream을 만들 때 사용한 방법이며 "**Base from member(c++ idioms)**"라는 이름을 가지고 있다.

2.3.2 Constructor and virtual function

가상함수를 호출하는 시점에 따라 Dervied 클래스의 함수를 호출하거나 Base의 함수를 호출하는 동작이 달라진다.

```
1 class Base{
2     public:
3     Base() { vfunc(); }    // Base vfunc
4     void foo() { vfunc(); } // Derived vfunc
5     virtual void vfunc() { cout << "Base vfunc()" << endl; }
6 };
7
8 class Derived : public Base {
9     int data{10};
10    public:
11    virtual void vfunc() override{ cout << "Derived vfunc()" << data << endl; }
12 };
13
14 int main() {
15     Derived d;
16     d.foo(); // Derived의 vfunc가 실행된다.
17 }
```

생성자에서는 가상 함수가 동작하지 않는다. 왜 이렇게 설계되었을까? Dervied 생성자는 컴파일러를 통해 다음과 같이 생성된다.

```
1 class Derived : public Base {
2     int data{10};
3     public:
4     Derived() : Base(), data(10)
5     virtual void vfunc() override{ cout << "Derived vfunc()" << data << endl; }
6 }
```

Base()가 먼저 호출되기 때문에 data의 정보를 사용하는 Derived의 vfunc를 호출하면 잘못된 정보를 호출하게 된다. 따라서 Base 생성자에서는 Base vfunc가 호출된다.

2.4 This call

```

1  class Point{
2      int x{0};
3      int y{0};
4  public:
5      void set(int a, int b) {
6          x=a; y=b;
7      }
8  };
9
10 int main(){
11     Point pt1;
12     Point pt2;
13
14     pt1.set(10,20);
15     pt2.set(10,20);
16 }
```

pt1, pt2를 생성하면 메모리 공간에 **멤버 변수가 객체 당 하나씩 생성된다.** 그렇다면 멤버 함수 set 또한 객체 당 하나씩 생성될까? 그렇지 않다. 멤버 함수는 객체가 여러개 생성되어도 **메모리에 한 개만** 만들어져 있다.

그렇다면 set 함수 인자가 a, b 밖에 없는데 x가 어떤 객체의 멤버인지 어떻게 알 수 있을까? 컴파일러가 이런 문제를 해결하기 위해 다음과 같이 컴파일 해준다.

```

1  class Point{
2      int x{0};
3      int y{0};
4  public:
5      void set(Point* this, int a, int b) {
6          this->x=a; this->y=b;
7      }
8  };
9  int main(){
10     Point pt1;
11     Point pt2;
12
13     set(pt1, 10,20);
14     set(pt2, 10,20);
15 }
```

사용자가 2개의 인자로 set를 구성했다하더라도 컴파일러가 객체 포인터 주소 인자가 같이 전달되도록 3개의 인자로 구성된 set 함수가 완성된다. 이러한 기술을 **this call**이라고 한다.

주의할 점은 실제 함수 인자가 전달되는 방식과 객체 주소가 전달되는 방식은 어셈블리 레벨에서는 약간 차이가 있다. 컴파일러마다 구현하는 방식도 다르니 위 예제 코드는 저런 개념으로 전달된다 정도만 숙지하면 된다.

static 멤버 함수는 this call이 적용되지 않는다.

```

1  class Point{
2      int x{0};
3      int y{0};
4  public:
5      void set(int a, int b) {
6          x=a; y=b;
7      }
8
9      static void foo(int a) {
10         x=a;
11     }
12 };
13
14 int main(){
15     Point pt1;
16     Point pt2;
17
18     pt1.set(10,20);
19     pt2.set(10,20);
20
21     Point::foo(10); // static 멤버함수는 객체 주소를 전달하지 않는다.
```

```
22     pt1.foo(10); // 실제로는 객체 주소 없이 Point::foo(10)으로 변환되어 실행된다.
23 }
```

2.5 Member function pointer

멤버 함수를 함수 포인터로 가르킬 수 있을까? 다음 예제를 보자.

```
1 class X {
2     public:
3     void mf1(int a) {}
4     static void mf2(int a) {}
5 };
6
7 void foo(int a) {}

8
9 int main() {
10     void(*f1)(int) = &foo;      // ok
11     void(*f2)(int) = &X::mf1; // error. this call에 의해 파라미터 개수 맞지 않음!
12     void(*f3)(int) = &X::mf2; // ok. static 멤버 함수는 this call이 없으므로 적용 가능
13 }
```

일반 함수 포인터에 **멤버 함수의 주소를 담을 수 없다**. 하지만 **static** 멤버 함수의 주소는 담을 수 있다. 그렇다면 멤버 함수의 주소를 담으려면 어떻게 해야 할까?

```
1 class X {
2     public:
3     void mf1(int a) {}
4     static void mf2(int a) {}
5 };
6
7 void foo(int a) {}

8
9 int main() {
10     void(*f1)(int) = &foo;
11     //void(*f2)(int) = &X::mf1;
12     void(*f3)(int) = &X::mf2;
13
14     void(X::*f2)(int) = &X::mf1; // ok. 멤버 함수 주소는 이렇게 담아야 한다.
15 }
```

일반 함수 포인터는 `void(*f1)(int) = foo` 또는 `&foo`를 입력해도 함수 주소로 암시적 변환이 가능하지만 멤버 함수 포인터는 `void(X::*f2) = X::mf1`으로 하면 컴파일 에러가 발생하므로 반드시 `&X::mf1`으로 주소를 넣어줘야 함에 유의한다.

멤버 함수 포인터는 일반 함수 포인터처럼 호출할 수 있을까?

```
1 f1(10); // ok
2 f2(10); // error. 객체가 필요하다.
```

멤버 함수 포인터는 객체가 있어야 사용 가능하다.

```
1 X obj;
2 obj.f2(10); // error. f2라는 멤버를 찾게 된다.
3
4 // pointer to member 연산자 사용
5 obj.*f2(10); // error. 연산자 우선순위 문제로 괄호가 먼저 계산된다.
6 (obj.*f2)(10); // ok
```

`.*`는 하나의 연산자 역할을 하며 **pointer to member operator**라고 부른다. 객체가 포인터인 경우 `->*`를 사용하면 된다.

```
1 (obj.*f2)(10); // ok
2 (pobj->*f2)(10); // ok
```

2.5.1 std::invoke, std::mem_fn

멤버 함수 포인터를 사용하는 형태가 너무 복잡해보인다. 이를 일반 함수 포인터처럼 조금 더 쉽게 호출할 수는 없을까?

```
1 class X {
2     public:
3         void mf1(int a) {}
4         static void mf2(int a) {}
5     };
6
7     void foo(int a) {}
8
9     int main() {
10         void(*f1)(int) = &foo;
11         void(X::*f2)(int) = &X::mf1;
12
13         X obj;
14
15         f1(10);           // 일반 함수 포인터 사용
16         (obj.*f2)(10); // 멤버 함수 포인터 사용
17         f2(&obj, 10); // 위 모양이 너무 복잡해서 이렇게 사용할 수는 없을까? 실제 논의가 있었다.
18                         // 이를 uniform call syntax라고 한다.
19 }
```

하지만 **uniform call syntax**는 컴파일러가 어셈블리 레벨을 많이 바꿔야 하기 때문에 채택되지 않았다. 대신 STL에서는 몇 가지 도구를 제공한다.

std::invoke는 c++17부터 나왔으며 일반 함수 포인터와 멤버 함수 포인터(정확히는 **callable object**)를 정확히 동일한 방법으로 호출할 수 있다.

```
1 #include <functional>
2
3 std::invoke(f1, 10);
4 std::invoke(f2, obj, 10);
5 std::invoke(f2, &obj, 10);
```

또는 **std::mem_fn**을 사용해도 된다. 이는 c++11부터 나왔으며 멤버 함수 주소를 인자로 받아서 함수 주소를 담은 래퍼 객체를 반환한다.

```
1 #include <functional>
2
3 auto f3 = std::mem_fn(&X::mf1); // 반드시 auto로 받아야 한다.
4 f3(obj, 10);
5 f3(&obj, 10);
```

2.6 Member data pointer

일반 변수의 포인터처럼 멤버 변수의 포인터도 만들 수 있을까?

```
1 struct Point {
2     int x;
3     int y;
4 };
5
6 int main() {
7     int num=0;
8     int *p1 = &num; // ok
9
10    int Point::*p2 = &Point::y; // ok. 멤버 변수의 포인터는 이렇게 가르켜야 한다.
11 }
```

p1은 num 변수의 메모리 주소를 가지고 있다. 그런데 Point 타입의 객체가 존재하지 않는데 **p2는 무엇을 담고 있을까?** 대부분의 컴파일러는 **Point 구조체 안에서 y의 offset을 담고 있다.** 이는 공식 표준은 아니지만 대부분의 컴파일러가 이렇게 구현되어 있다.

멤버 변수 포인터는 다음과 같이 사용한다.

```
1 *p1 = 10; // ok
2 *p2 = 10; // error
3
4 Point pt;
5 pt.*p2 = 10; // ok. pt.y = 10
6 // *(&pt + p2) = 10; p2만큼 오프셋 주소를 더하여 그 곳의 변수에 10을 넣는다.
```

멤버 변수 포인터도 invoke를 통해 호출할 수 있다.

```
1 std::invoke(p, obj) = 10; // ok. obj.y = 10. 일반 함수 포인터처럼 사용할 수 있다.
2 int n = std::invoke(p, obj); // ok. 데이터를 꺼내는 것도 가능하다
```

std:invoke는 일반 포인터, 함수, 멤버 함수 포인터, 멤버 변수 포인터, 람다 표현식을 전부 일관되게 동작할 수 있게 해준다. 이런 타입들을 **callable type**라고 한다.

2.7 Implement custom max

이번 섹션에서는 임의의 max 함수를 작성해보자.

```
1 #include<iostream>
2 #include<string>
3
4 template<class T>
5 const T& mymax(const T& obj1, const T& obj2) {
6     return obj1 < obj2 ? obj2 : obj1;
7 }
8
9 int main() {
10     std::string s1 = "abcd";
11     std::string s2 = "xyz";
12
13     auto ret1 = mymax(s1, s2);
14     std::cout << ret1 << std::endl;
15 }
```

mymax 함수는 문자열의 처음 글자인 a와 x를 비교하여 x가 더 뒤에 있으므로 s2를 반환하게 된다. **만약 문자열의 순서가 아니라 문자열의 개수를 mymax를 통해 비교하고 싶다면 어떻게 해야 할까?**

이와 같이 알고리즘 함수가 사용하는 "정책(비교 방식)"을 변경하고 싶은 경우 다음과 같은 방법을 사용 할 수 있다.

- 이항 조건자(binary predicator) 사용 (e.g., c++ stl)
-

```
1 std::sort(v.begin(), v.end(), [](auto &a, auto &b) { return a.size() < b.size(); });
```

- 단항 조건자(unary predicator) 사용 (e.g., python)
-

```
1 sorted(str_list, key = lambda x : len(x));
```

- 단항, 다항 조건자를 모두 사용 (e.g., c++ 20 range 알고리즘의 원리). 멤버 함수 포인터, 멤버 데이터 포인터, std:invoke를 모두 활용한다.

```
1 template<class T, class Proj>
2 const T& mymax(const T& obj1, const T& obj2, Proj proj) {
3     return proj(obj1) < proj(obj2) ? obj2 : obj1; // proj()를 통해 비교하고 결과값은 원래 값을 반환한다.
4 }
5
6 int main() {
7     std::string s1 = "abcd";
8     std::string s2 = "xyz";
9
10    auto ret1 = mymax(s1, s2, [](auto &a) { return a.size(); });
11    std::cout << ret1 << std::endl;
```

12 }

mymax의 3번째 인자로 단항 조건자를 전달하면 비교 시 조건자의 결과를 비교한다. 이는 c++20에서 **Projection**이라는 기술로 불린다.

만약 단항 조건자 대신 멤버 함수 포인터를 전달할 수는 없을까?

```
1 template<class T, class Proj>
2 const T& mymax(const T& obj1, const T& obj2, Proj proj) {
3     return proj(obj1) < proj(obj2) ? obj2 : obj1;
4 }
5
6 int main() {
7     std::string s1 = "abcd";
8     std::string s2 = "xyz";
9
10    auto ret1 = mymax(s1, s2, [](auto &a) { return a.size(); });
11    auto ret2 = mymax(s1, s2, &std::string::size ); // error. 멤버 함수 포인터이므로 proj(obj)에서 에러가
12        발생한다.
13 }
```

proj(obj)는 일반 함수라면 ok이지만 멤버 함수라면 에러가 발생한다. 이 때, **std::invoke**를 사용하면 일반함수와 멤버 함수 모두 커버할 수 있다.

```
1 template<class T, class Proj>
2 const T& mymax(const T& obj1, const T& obj2, Proj proj) {
3     return std::invoke(proj, obj1) < std::invoke(proj, obj2) ? obj2 : obj1; // 일반 함수, 멤버 함수 모두
4         커버 가능
5 }
6
7 int main() {
8     std::string s1 = "abcd";
9     std::string s2 = "xyz";
10
11    auto ret1 = mymax(s1, s2, [](auto &a) { return a.size(); }); // ok
12    auto ret2 = mymax(s1, s2, &std::string::size ); // ok.
13 }
```

Projection은 생략될 수 있어야 한다. 이를 위해서는 Proj의 디폴트 값이 있어야 한다. **std::identity**는 전달 받은 인자를 어떤 변화 없이 참조값을 반환하는 함수 객체이다. 이를 활용하여 디폴트 값을 정의한다.

```
1 template<class T, class Proj = std::identity> // std::identity로 디폴트 값 설정
2 const T& mymax(const T& obj1, const T& obj2, Proj proj = {}) {
3     return std::invoke(proj, obj1) < std::invoke(proj, obj2) ? obj2 : obj1;
4 }
5
6 int main() {
7     std::string s1 = "abcd";
8     std::string s2 = "xyz"
9
10    auto ret1 = mymax(s1, s2, [](auto &a) { return a.size(); }); // ok
11    auto ret2 = mymax(s1, s2, &std::string::size ); // ok.
12    auto ret3 = mymax(s1, s2); // ok
13 }
```

std::identity은 c++20에 처음 등장하였으므로 구현 코드는 다음과 같이 되어 있다. <functional> 헤더 파일이 필요하다.

```
1 struct identity {
2     template<class _Ty>
3     [[nodiscard]] constexpr
4     _Ty&& operator() (_Ty&& arg) const noexcept {
5         return std::forward<_Ty>(arg);
6     }
7     using is_transparent = int;
8 }
```

다음으로 mymax를 통해 임의의 객체 Point를 비교해보자.

```
1 template<class T, class Proj = std::identity>
2 const T& mymax(const T& obj1, const T& obj2, Proj proj = {}) {
3     return std::invoke(proj, obj1) < std::invoke(proj, obj2) ? obj2 : obj1;
4         // (obj1.*proj) < (obj2.*proj)
5 }
6
7 struct Point {
8     int x,y;
9 }
10
11 int main() {
12     Point p1 = {2,0};
13     Point p2 = {1,1};
14
15     auto ret = mymax(p1, p2, &Point::y); // 이렇게 비교할 수는 없을까?
16     cout << ret.x << ", " << ret.y << endl;
17 }
```

std::invoke는 멤버 변수의 포인터도 커버할 수 있다. 따라서 위 코드는 아무 수정 없이 정상적으로 동작한다. 정리하면 mymax는 다음과 같이 4가지 사용법이 존재한다.

```
1 string s1 = "abcd";
2 string s2 = "xyz";
3
4 auto ret1 = mymax(s1, s2); // (1)
5 auto ret2 = mymax(s1, s2, [](auto &a) { return a.size(); }); // (2)
6 auto ret3 = mymax(s1, s2, &std::string::size); // (3)
7
8 Point p1 = {0,0};
9 Point p2 = {1,1};
10
11 auto ret4 = mymax(p1, p2, &Point::y); // (4)
```

2.7.1 Add compare operator

앞서 살펴본 mymax 함수는 std::invoke를 사용하여 여러 케이스에 대해 사용 가능한 훌륭한 함수였다.

```
1 template<class T, class Proj = std::identity>
2 const T& mymax(const T& obj1, const T& obj2, Proj proj = {}) {
3     return std::invoke(proj, obj1) < std::invoke(proj, obj2) ? obj2 : obj1;
4 }
```

mymax 함수는 < 연산을 기본으로 하고 있다. 이를 유저가 원하는 임의의 비교 연산자를 사용할 수는 없을까? 세번째 인자에 Comp 연산자를 추가함으로서 이를 가능하게 할 수 있다.

```
1 template<class T, class Comp = std::less<void>, class Proj = std::identity>
2 const T& mymax(const T& obj1, const T& obj2, Comp comp = {}, Proj proj = {}) {
3     return std::invoke(comp, std::invoke(proj, obj1), std::invoke(proj, obj2)) ? obj2 : obj1;
4         // comp가 멤버 함수일 때는 comp(a,b)가 불가능하기 때문에 std::invoke(comp, a, b)로 작성해준다.
5 }
6
7 int main(){
8     std::string s1 = "abcd";
9     std::string s2 = "xyz";
10
11     auto ret1 = mymax(s1, s2);
12     auto ret2 = mymax(s1, s2, std::greater{});
13     auto ret2 = mymax(s1, s2, {}, &std::string::size); // 세번째 {}는 디폴트 연산자를 사용하라는 의미
14     auto ret2 = mymax(s1, s2, std::greater{}, &std::string::size);
15 }
```

왜 템플릿의 디폴트 인자를 std::less<T>가 아닌 std::less<void>로 했을까? 이는 callable section을 공부하고 오면 답을 알 수 있다.

2.7.2 c++20 ranges algorithm

지금까지 구현한 mymax는 c++20에서 도입된 range algorithm 기반의 std::ranges::max 함수와 거의 동일한 구현 코드를 갖는다. 하지만 이번 섹션에서 만든 **mymax는 함수 템플릿이지만 std::ranges::max는 함수 객체(템플릿)**임에 유의한다.

```
1 template<class T, class Comp = std::less<void>, class Proj = std::identity>
2 const T& mymax(const T& obj1, const T& obj2, Comp comp = {}, Proj proj = {}) {
3     return std::invoke(comp, std::invoke(proj, obj1), std::invoke(proj, obj2)) ? obj2 : obj1;
4 }
5
6 int main(){
7     std::string s1 = "abcd";
8     std::string s2 = "xyz";
9
10    auto ret1 = mymax(s1, s2);
11    auto ret2 = mymax(s1, s2, std::greater{}, &std::string::size);
12    auto ret3 = std::ranges::max(s1, s2, std::ranges::greater, &std::string::size); // ret2와 동일한
13                                결과 출력
}
```

c++20의 range algorithm은 알고리즘의 "비교 정책"을 교체할 수 있으며 "Projection"을 전달할 수 있다. 이 때 std::invoke를 통해 구현하여 멤버 함수 포인터와 멤버 데이터 포인터 모두 사용이 가능하다. 또한 반복자 구간이 아닌 컨테이너를 전달받기 때문에 다음과 같이 조금 더 편하게 코딩이 가능하다.

```
1 std::vector<std::string> v = {"hello", "a", "xxx", "zz"};
2
3 std::sort(v.begin(), v.end()); // 매 번 begin, end를 적어줘야 하므로 불편함
4
5 // c++20 ranges
6 std::ranges::sort(v); // 컨테이너만 넘겨주면 자동으로 정렬. 디폴트로 알파벳 순서대로 정렬
7 std::ranges::sort(v, std::ranges::greater{}, &std::string::size); // 비교 정책과 Projection 모두 전달 가능
```

2.8 Size of member function pointer

이번 섹션에서는 멤버 함수 포인터의 크기를 알아본다. 이를 알기 위해서는 우선 다중상속을 정확히 이해하고 있어야 한다.

```
1 struct A {int x;};
2 struct B {int y;};
3 struct C : public A, public B { // 다중 상속
4     int z;
5 };
6
7 int main(){
8     C cc;
9     cout << &cc << endl; // 1000
10
11    A* pA = &cc;
12    B* pB = &cc;
13    cout << pA << endl; // 1000
14    cout << pB << endl; // 1004. B클래스는 두번째로 상속을 받고 있으므로 A 클래스의 멤버 변수가 차지하는 4
15                                바이트를 건너뛰어 1004의 값을 갖는다
}
```

static_cast를 해도 똑같이 1004의 값이 나온다. reinterpret_cast를 사용해야 1000의 값이 나온다. **reinterpret_cast은 0x1000 위치의 메모리를 B타입처럼 사용하겠다라는 의미이다.** 메모리를 다르게(다른 타입으로) 사용하겠다는 의미이므로 사용에 주의해야 한다.

```
1 B* pB1 = static_cast<B*>(&cc);           // 1004
2 B* pB2 = reinterpret_cast<B*>(&cc) // 1000
```

다음으로 멤버 함수가 있는 예제를 보자.

```

1 struct A{
2     int x;
3     void fa() { std::cout << this << std::endl; }
4 };
5 struct B{
6     int y;
7     void fb() { std::cout << this << std::endl; }
8 };
9 struct C : struct A, struct B{
10     int z;
11     void fc() { std::cout << this << std::endl; }
12 };
13
14 int main() {
15     C cc;
16     cc.fc(); // 0x1000
17     cc.fa(); // 0x1000
18     cc.fb(); // 0x1004. 이전 코드와 동일하게 B에 대한 상속이 두번째로 받았으므로 메모리 공간에 4바이트만큼 뒤로
19     간 1004가 출력된다.
}

```

다음과 같이 함수 포인터가 있다고 하자.

```

1 void (C::*f)();
2
3 f = &C::fa;
4 (cc::*f)(); // 1000
5
6 f = &C::fb;
7 (cc::*f)(); // 1004. 정상적으로 함수 포인터가 동작한다.

```

함수 포인터는 런타임 시점에서 f(&cc)와 같이 호출된다. 하지만 f = &C::fb를 가리키는 경우 1004가 정상적으로 호출되려면 f(&cc + sizeof(A))가 호출되어야 한다. **컴파일 타임에선 f가 fa를 가리킬지 fb를 가리킬지는 알 수 없는데 어떻게 정상적으로 동작하는 것일까?**

멤버 함수 포인터는 항상 4byte(32bit), 8byte(64bit)인 것은 아니다. 코드에 따라 멤버 함수 포인터의 크기가 달라지는데 **다중 상속의 경우 함수 주소와 this offset을 같이 보관하여 8byte(32bit), 16byte(64bit)의 크기를 가진다.**

```

1 void(C::*f)(); // 함수 주소 + this offset을 함께 보관 8byte(32bit), 16bit(64bit)
2
3 f = &C::fa; // fa주소, 0이 보관
4 f = &C::fb; // fb주소, sizeof(A) 보관

```

위와 같은 동작이 c++ 표준 동작은 아니다. 컴파일러마다 원리가 다를 수 있으나 대부분의 컴파일러가 위와 같은 방법을 통해 멤버 함수 포인터를 관리한다.

Tip

void*가 모든 주소를 담는다고 알려져 있지만 엄밀하게 보면 {주소, this offset}을 보관하는 **멤버 함수의 포인터는 담을 수 없다.** 또한 멤버 데이터의 포인터도 진짜 메모리 주소가 아닌 상대적인 offset 정보만 가지고 있기 때문에 **멤버 데이터 포인터 또한 담을 수 없다.**

```

1 struct myostream{
2     myostream& operator<<(int n) { printf("int : %d\n", n); return *this; }
3     myostream& operator<<(double d) { printf("double : %d\n", n); return *this; }
4     myostream& operator<<(bool b) { printf("bool %d\n", n); return *this; }
5     myostream& operator<<(void* p) { printf("void*: %d\n", n); return *this; }
6 }
7 myostream mycout;
8
9 int main() {
10     int n=10;
11     double d=3.4;

```

```

12     int Point::*p = &Point::y;
13
14     mycout << n; // int
15     mycout << d; // double
16     mycout << &n; // void*
17     mycout << &d; // void*
18     mycout << p; // 1. 1이 출력되는 이유는 void*로 변환될 수 없는 멤버 변수 포인터가 bool로는 암시적 변환이
19         되기 때문에 4 -> 1(true)가 되어 1이 출력되게 되는 것이다.
}

```

따라서 멤버 함수의 포인터를 출력해보고 싶을 때는 cout보단 printf를 써서 출력하는 것이 좋다.

2.9 new, delete and placement new

이번 섹션에서는 new, delete 키워드에 대해서 자세히 살펴본다.

```

1 class Point{
2     int x,y;
3     public:
4     Point(int a, int b) : x{a}, y{b} { cout << "Point(int,int)" << endl; }
5     ~Point() { cout << "~Point()" << endl; }
6 }
7
8 int main(){
9     Point* p1 = new Point(1,2); // 메모리 할당, 생성자 호출
10    delete p1; // 소멸자 호출, 메모리 해제
11 }

```

c++에서 new 키워드를 사용하면 다음과 같은 메모리 할당, 생성자 호출 코드가 자동으로 수행된다.

```

1 Point* p1 = new Point(1,2);
2
3 void* p = operator new(sizeof(Point)); // 1. 메모리 할당
4 Point* p1 = new(p) Point(1,2); // 2. 생성자 호출. new(p)를 placement new라고 부른다.

```

delete를 사용하면 소멸자가 호출되고 메모리가 해제되는 코드가 자동으로 수행된다.

```

1 delete p1;
2
3 p1->~Point(); // 1. 소멸자 호출
4 operator delete(p1); // 2. 메모리 해제

```

만약 생성자 호출 없이 메모리만 할당, 해제하고 싶은 경우 다음과 같이 쓰면 된다. c언어에서 malloc과 free한 것과 동일하다.

```

1 void* p = operator new(sizeof(Point)); // Point 구조체는 x,y로 인해 8바이트 할당
2
3 operator delete(p); // 메모리 해제

```

operator new, delete 함수는 c++20 기준으로 각각 22개, 30개의 다른 버전이 있다. 두 함수는 <new> 헤더를 필요로 하며 std namespace가 아닌 global namespace에 존재한다.

```

1 [[nodiscard]] void* operator new(std::size_t);
2 void operator delete (void* ptr) noexcept;

```

앞서 메모리를 할당한 다음 생성자를 호출하려면 아래와 같이 하면 된다.

```

1 void *p1 = operator new(sizeof(Point));
2 Point *p2 = new(p1) Point(1,2); // 생성자 호출

```

new(p1)은 **placement new**라고 불리며 메모리 할당없이 이미 할당된 메모리에 생성자를 명시적으로 호출하기 위한 new이다. c++20에서는 new(p) Point(1,2) 대신 **std::construct_at(p, 1, 2)**를 통해 생성자를 명시적으로 호출할 수 있다. 유사하게 **std::destroy_at(p)**를 통해 소멸자를 호출할 수 있다.

void*가 아닌 정확한 클래스 타입으로 메모리를 할당받기 위해서는 다음과 같이 작성한다.

Tip

- new Point(1,2); // 새로운 메모리를 할당하고 객체 생성
- new(p) Point(1,2); // 이미 할당된 메모리(p)에 객체 생성 (=placement new)
- std::construct_at(p, 1, 2); // 위와 동일. c++20, <memory>
- std::destroy_at(p); // 명시적 소멸자 호출

```
1 Point* p3 = static_cast<Point*>(operator new(sizeof(Point)));
2 std::construct_at(p3, 1, 2); // 생성자 호출
```

2.9.1 Using placement new

메모리 할당과 생성자 호출을 분리해야 하는 이유가 있을까?

```
1 class Point{
2     int x,y;
3 public:
4     Point(int a, int b) :x{a}, y{b} {} // 디폴트 생성자가 없음에 주목!
5     ~Point() {}
6 };
7
8 int main(){
9     Point* p1 = new Point(0,0); // Point 객체 한개를 힙에 생성하고 싶은 경우
10
11    Point* p2 =? // 만약 Point 객체 3개를 연속적인 형태(배열 형태)로 생성하고 싶은 경우 어떻게 해야 할까?
12 }
```

p2를 new Point[3]와 같이 생성하면 Point 타입에는 반드시 디폴트 생성자가 있어야 한다. 하지만 **디폴트 생성자 없이도 3개를 연속적인 배열 형태로 만들고 싶은 경우도 발생할 수 있다.**

c++11에는 다음과 같이 초기화할 수 있다. {0,0}은 인자 2개짜리 생성자를 호출하기 때문에 에러없이 잘 빌드된다.

```
1 Point* p2 = new Point[3]{{0,0}, {0,0}, {0,0}};
```

하지만 3개가 아니라 30개, 100개가 넘어가는 경우는 어떻게 해야 할까? 이럴 때는 **메모리 할당과 생성자 호출을 분리하면 훨씬 편하게 생성할 수 있다.**

```
1 Point* p2 = static_cast<Point*>(operator new(sizeof(Point)*3));
2
3 for(int i=0; i<3; i++){
4     new(&p2[i]) Point(0,0); // c++20 이전 표기법
5     std::construct_at(&p2[i], 0,0); // c++20 이후 표기법
6 }
7
8 for(int i=0; i<3; i++){
9     p2[i].~Point(); // c++20 이전 표기법
10    std::destroy_at(&p2[i]); // c++20 이후 표기법
11 }
```

2.9.2 vector and placement new

```
1 int main(){
2     vector<int> v(10);
3
4     v.resize(7); // size의 크기를 줄이고 메모리 할당 크기는 capacity 변수에 저장한다.
5     cout << v.size() << endl; // 7
6     cout << v.capacity() << endl; // 10
```

```

7     v.resize(8);
8     cout << v.size() << endl; // 8
9     cout << v.capacity() << endl; // 10
10
11 }
```

벡터가 임의의 클래스 X를 담고 있다고 해보자.

```

1 struct X{
2     X() { cout << "X() get resource" << endl; }
3     ~X() { cout << "~X() release resource" << endl; }
4 }
5
6 int main(){
7     vector<X> v(10);
8
9     v.resize(7); // 크기가 줄어드는 경우 메모리는 제거하지 않더라도 소멸자는 불러야하지 않을까?
10
11    v.resize(8); // 8번째 메모리에 생성자가 호출되어야 하지 않을까?
12 }
```

사용자 정의 타입을 vector에 담으면 **메모리의 할당, 해지가 없는데도 불구하고 생성자, 소멸자의 호출을 정말 많이 호출**한다. 따라서 메모리 할당, 생성자 호출에 대한 개념의 정확한 이해가 필요하다.

2.10 Trivial constructor

2.10.1 Trivial default constructor

생성자가 trivial하다는 말은 컴파일러가 자동으로 생성해주면서 동시에 아무 일도 하지 않을 때를 말한다

2.10.2 Trivial copy constructor

복사 생성자가 trivial하다는 말은 멤버변수 값은 복사하는 것 이외에 아무 일도 하지 않을 때를 말한다. 복사 생성자가 trivial하다면 배열 전체를 memcpy와 memmove 등으로 복사하는 것이 빠르다! 복사 생성자가 trivial하지 않다면 배열의 모든 요소에 대해 하나씩 “복사 생성자”를 호출해서 생성자를 호출해야 한다

```

1 struct Point {
2     int x=0;
3     int y=0;
4 };
5
6 template<class T>
7 void constexpr copy_type(T* dst, T* src, std::size_t sz) {
8     if(std::is_trivially_copy_constructible_v<T>) {
9         std::cout << "using memcpy" << std::endl;
10        memcpy(dst, src, sizeof(T)*sz);
11    }
12    else {
13        std::cout << "using copy ctor" << std::endl;
14        while(sz--){
15            new(dst) T(*src);
16            --dst, --src;
17        }
18    }
19 }
20
21 int main(){}
22 Point arr1[5];
23 Point arr2[5];
24 copy_type(arr1, arr2, 5);
25 }
```

위 코드에서 Point 클래스는 int x,y와 같이 간단한 멤버변수만 존재하므로 trivial copy constructor이다. 하지만 virtual void foo() 같이 가상함수를 사용하거나 string s;와 같이 복사하는 클래스를 사용하게 되면 trivial하지 않게 된다. 이런 경우에는 **placement new** 또는 **std::construct_at**을 사용해야 한다.

2.11 Type deduction

컴파일 타입에 타입이 결정되는 auto 키워드에 대해 살펴보자

```
1 int main(){
2     int n=10;
3     const int c =10;
4
5     auto a1 = n; // int a1=n;
6     auto a2 = c; // (1) const int a2 = c; --> no.
7         // (2) int a2 = c; --> ok.
8 }
```

type deduction(타입 추론)이 발생하는 키워드는 다음과 같다: template, auto, decltype

```
1 #include <iostream>
2 template<class T> void foo(T arg){
3     std::cout << typeid(T).name() << std::endl;
4 }
5 int main(){
6     int n=10;
7     foo(n);           // T=int
8     foo<const int&>(n); // T=const int&. But typeid(T).name() keep printing output 'int'
9 }
```

typeid(T).name()은 타입 이름만 추론할 뿐 const, volatile, reference 정보가 출력되지 않는다.

1. 이럴 때는 의도적으로 예러를 발생시켜서 정확한 타입을 예러 메시지를 통해 알 수 있다.
2. 또는 boost::type_index 라이브러리에 type_id_with_cvr<T>().pretty_name()을 사용하면 됨
3. 컴파일러가 제공하는 매크로를 사용하면 된다.
 - __FUNCTION__: 함수의 이름만 보여주므로 타입 추론에는 사용하지 않음
 - __PRETTY_FUNCTION__: g++, clang에서는 함수 이름과 타입이 나옴
 - __FUNCSIG__: cl.exe 버전

```
1 std::cout << __FUNCTION__ << std::endl;
2 std::cout << __PRETTY_FUNCTION__ << std::endl; // g++, clang
3 std::cout << __FUNCSIG__ << std::endl;          // cl.exe
```

T가 값인 경우를 살펴보자

```
1 #include <iostream>
2 template<class T> void foo(T arg){
3     while(--arg>0) {}
4 }
5
6 int main(){
7     int n=10;
8     int& r = n;
9     const int c = 10;
10    const int& cr = c;
11    foo(n); // T=int
12    foo(r); // T=int& 일 것 같지만 T=int
13    foo(c); // T=const int 일 것 같지만 T=int
14    foo(cr); // T=const int& 일 것 같지만 T=int
15 }
```

T 인자를 값으로 받을 때는 복사본 객체가 만들어져서 “const, volatile, reference” 속성을 제거하고 값만 받는다. 헷갈리는 것 중 하나가 값으로 받을 때는 인자의 const 속성을 제거되고 “인자가 가리키는 곳의 const 속성을 유지”한다. 무슨 이야기인지 살펴보자

```
1 #include <iostream>
2 template<class T> void foo(T arg){
3     std::cout << __PRETTY_FUNCTION__ << std::endl;
```

```

4 }
5 int main(){
6     const char* const s = "hello";
7     foo(s); // 포인터의 const 속성은 제거되지만 가리키는 곳 "hello"의 const 속성은 유지됨!
8         // const char* arg = "hello"가 됨!!
9 }
```

T 인자를 참조로 받을 때는 다음과 같다.

```

1 #include <iostream>
2 template<class T> void foo(T& arg){
3     std::cout << __PRETTY_FUNCTION__ << std::endl;
4 }
5
6 int main(){
7     int n = 10;           // T=int.      arg=int&
8     int& r = n;          // T=const int. arg=const int& (const 속성은 유지!)
9     const int c = 10;    // T=int      arg=int&. (T에서 reference 속성을 제거!)
10    const int& cr = c; // T=const int  arg=const int& (const 속성은 유지!)
11 }
```

주의해야 할 점은 T의 타입과 arg의 타입은 다르다는 것이다 (T& arg 이기 때문!). 함수 인자의 “reference를 제거하고 T의 타입을 결정한다”. 인자가 가진 “const, volatile 속성을 유지한다.”. 마지막으로 T에 배열이 전달된 경우를 살펴보자

```

1 template<class T> void foo(T arg){
2     std::cout << __PRETTY_FUNCTION__ << std::endl;
3 }
4 template<class T> void goo(T& arg){
5     std::cout << __PRETTY_FUNCTION__ << std::endl;
6 }
7
8 int main(){
9     int x[3] = {1,2,3};
10    foo(x); // T=int* 타입으로 받는다
11    goo(x); // T=int[3] 타입으로 받는다. arg는 int() [3] 타입으로 받는다
12 }
```

T& arg로 배열을 받는 경우 $\&arg[3] = x$; 처럼 받는게 되어서 배열의 reference가 된다. 따라서 아래와 같이 goo()를 사용하면 에러가 발생한다.

```

1 template<class T> void foo(T arg){
2     std::cout << __PRETTY_FUNCTION__ << std::endl;
3 }
4 template<class T> void goo(T& arg){
5     std::cout << __PRETTY_FUNCTION__ << std::endl;
6 }
7
8 int main(){
9     foo("orange", "apple"); // ok
10    goo("orange", "apple"); // error
11 }
```

foo와 **goo** 둘 다 const char 형식으로 받지만 포인터는 개수에 제한이 없으므로 ok인 반면에 reference는 const char[7], const char[6]은 다른 reference이기 때문에 같은 T를 받는 상황에서 에러가 발생한다!
foo(const char[7], const char[6]), **goo**(const char[7], const char[6])

2.11.1 Auto type deduction

template은 함수 인자로 추론하는 반면 **auto**는 우변의 표현식으로 타입을 추론한다. template을 ‘T arg = 함수 인자’처럼 추론한다고 볼 수 있으므로 사실 ‘auto a = 표현식’과 동일한 형태로 추론한다!

```

1 int main(){
2     int n=10;
3     int& r = n;
```

```

4 const int c = 10;
5 const int& cr = c;
6
7 auto a1 = n; // auto=int
8 auto a2 = r; // auto=int
9 auto a3 = c; // auto=int
10 auto a4 = cr; // auto=int
11
12 auto& a5 = n; // auto=int. a5=int&
13 auto& a6 = r; // auto=int. a6=int&
14 auto& a7 = c; // auto=const int. a7=const int&
15 auto& a8 = cr; // auto=const int. a8=const int&
16
17 int x[3] = {1,2,3};
18 auto a = x; // auto=int*
19 auto& b = x; // auto=int[3]. b=int(&)[3]
20 }

```

위와 같이 template에서 본 규칙들이 그대로 적용! 아래와 같이 조금 더 까다로운 경우를 보자.

```

1 int main(){
2     auto a1 = 1;
3     auto a2 = {1};
4     auto a3{1};
5
6     std::cout << typeid(a1).name() << std::endl; // auto=int
7     std::cout << typeid(a2).name() << std::endl; // auto=initialized_list
8     std::cout << typeid(a3).name() << std::endl; // auto=int
9     std::vector<int> v1(10,0);
10    std::vector<bool> v2(10,false);
11
12    auto a4 = v1[0];
13    auto a5 = v2[0];
14
15    std::cout << typeid(a4).name() << std::endl; // auto=int
16    std::cout << typeid(a5).name() << std::endl; // auto=temporary proxy 객체
17 }

```

배열은 auto a= 1라고 하면 배열 타입으로 추론된다. bool 타입은 최적화 과정에서 specialization되어 있다. 따라서 bool은 [] 연산자가 bool로 변환 가능한 **temporary proxy**으로 추론한다!

2.11.2 Array Name

배열의 이름은 배열의 1번째 요소의 주소로 암시적 형변환 된다.

```

1 int main(){
2     int x[3] = {1,2,3};
3
4     int *p0[3] = &x; // error. 연산자 우선 순위에 따라 p[3], *p 순으로 추론된다.
5     int (*p1)[3] = &x; // ok. (*p)로 감싸줘야 x[3] 배열에 대한 제대로된 포인터가 된다.
6     int *p2 = x; // ok. &x[0]
7
8     printf("%p, %p\n", p1, p1+1); // 배열 자체를 가리키므로 +1을 하면 12바이트만큼 증가한다.
9     printf("%p, %p\n", p2, p2+1); // 배열의 첫번째 원소를 가리키므로 +1을 하면 4바이트만큼 증가한다.
10
11    (*p1)[0] = 10;
12    *p2 = 10;
13 }

```

위 코드에서 보면 p1, p2가 가리키는 주소는 동일하지만 포인터 타입이 다르므로 주소를 해석하는 방식이 다르다. 다음과 같이 배열을 함수 인자를 받는 경우에 대해 알아보자

```

1 void f1(int p[3]) {
2     printf("%d\n", sizeof(p)); // 마치 3개 배열을 입력으로 받는 듯 보이지만 int *p와 동일한 포인터를 받고
3     // 있다. 즉, 12가 아닌 8(64bit)이 나온다
4 }

```

```

4
5 int main(){
6     int x[3] = {1,2,3};
7     f1(x);
8 }
```

함수 인자로 배열을 받을 때는 int *p와 같이 포인터 타입으로 받거나 int p[]와 같이 배열 타입으로 받는다. 이 때 컴파일러에 의해 둘 다 **포인터**로 변환한다. (int *p).

int p[3]와 같이 배열의 크기도 지정해줄 수 있는데 이 또한 **포인터**로 컴파일러가 변환한다(int *p). 헷갈리기 쉬운 문법이니 주의한다. 마지막으로 다차원 배열의 포인터에 대해 알아보자

```

1 void foo(int (*p)[2]) {
2     p[0][0] = 100;
3 }
4 int main(){
5     int y[3][2] = {1,2,3,4,5,6};
6
7     int (*p3)[3][2] = &y; // 배열의 변수와 정확히 동일한 형태를 유지하고 (*p)로 감싸주면 배열 포인터가 된다.
8     int (*p4)[2] = y;    // 배열의 첫번째 원소는 2차원 배열이므로 (=1,2) int (*p4)[2]와 같이 선언해줘야
9     한다.
10
11     foo(y);
12 }
```

2.12 Rvalue & forwarding & reference

2.12.1 Lvalue vs Rvalue

- lvalue: 등호의 왼쪽에 올 수 있는 표현식
- rvalue: 등호의 왼쪽에 올 수 없는 표현식

이외에도 c++에서는 다음과 같은 추가적인 구분 방법이 존재한다.

```

1 x=10;
2 int f1() { return x; }
3 int& f2() { return x; }
4
5 int main(){
6     int v1=0, v2=0;
7
8     v1=10; // ok. v1 : lvalue
9     10=v1; // error. 10 : rvalue
10    v2=v1;
11    int *p1 = &v1; // ok
12    int *p2 = &10; // error
13
14    f1() = 20; // error
15    f2() = 20; // ok
16    const int c = 10; // 상수도 lvalue의 특징을 가지고 있다 (이름, 주소)
17    c = 20; // error
18    "aa"[0] = 'x'; // error. lvalue 문제가 아니라
                      // const char[3]이므로 에러 발생!
19
20 }
```

이름, 주소가 있으면 lvalue, 없으면 rvalue. 참조를 반환하면 lvalue, 값을 반환하면 rvalue라고 볼 수 있다. 다음과 같은 의문이 들 수 있다.

1. **모든 상수는 rvalue인가?** : 상수는 immutable lvalue로 취급한다.
2. **모든 rvalue는 상수인가?** : Point(1,2).set(10,20) 같이 temporary 객체도 멤버 변수를 호출할 수 있으므로 상수가 아니다

lvalue, rvalue에 대한 흔한 오해 중 하나가 객체, 변수에 부여되는 속성으로 오해한다. **하지만 이는 표현식 (expression)에 부여되는 속성이다!** 표현식이란 “하나의 값”을 만들어내는 코드 집합을 말한다!

```

1 int main(){
2     int n=3;
3
4     n=10;      // ok
5     n+2 = 10; // error. n+2=5인데 이는 값이므로 rvalue!
6     n+2*3 = 10; // error.
7
8     (n=20) = 10; // ok. 표현식 ok
9
10    ++n = 10; // ok
11    n++ = 10; // error. n=3-->4로 바뀌는데 이는 값이므로 error
12 }
```

어떤 값이 lvalue인지 rvalue인지 조사하려면 **decltype**을 사용하면 된다!

```

1 int main(){
2     int n = 10;
3
4     if(std::is_lvalue_reference_v<decltype(n++)>) // n++: rvalue, ++n: lvalue
5         std::cout << "lvalue" << std::endl;
6     else
7         std::cout << "rvalue" << std::endl;
8
9     if(std::is_lvalue_reference_v<decltype((n))>) // 그냥 n을 넣으면 rvalue로 잘못나온다. (n)을 통해
10        괄호로 감싸줘야 표현식으로 인식해서 lvalue로 정상 인식한다!
11        std::cout << "lvalue" << std::endl;
12    else
13        std::cout << "rvalue" << std::endl;
14 }
```

다음과 같이 매크로를 만들어 놓으면 편하게 구분할 수 있다.

```

1 #define value_category(...)
2     if( std::is_lvalue_reference_v<decltype((__VA_ARGS__))> )
3         std::cout << "lvalue" << std::endl;
4     else if( std::is_rvalue_reference_v<decltype((__VA_ARGS__))> )
5         std::cout << "rvalue(xvalue)" << std::endl;
6     else
7         std::cout << "rvalue(prvalue)" << std::endl;
8
9 int main(){
10     int n=10;
11
12     value_category(n);
13     value_category(n+2);
14     value_category(n++);
15     value_category(++n);
16 }
```

2.12.2 Reference & Overloading

참조자(reference)의 규칙에 대해 알아보자

```

1 int main(){
2     int n=3;
3
4     int& r1 = n; // ok
5     int& r2 = 3; // error
6     const int& r3 = n; // ok
7     const int& r4 = 3; // ok (하지만 상수성이 추가됨)
8
9     int&& r5 = n; // error
10    int&& r6 = 3; // ok. (상수 성질 없음! rvalue reference라고 부름)
11 }
```

임의의 숫자를 가리키고 싶을 땐 const int& r4 = 3과 같이 가리켜야 했으나 c++11 이후 rvalue-reference라는 문법이 등장하면서 int&& r6=3과 같이 가리킬 수 있게 되었음.

기존 int& r1을 **lvalue-reference**라고 부르며 int&& r6=3을 **rvalue-reference**라고 부름! 그렇다면 왜 상수성 없이 rvalue를 가리키는 것이 중요할까? 이는 move semantics와 perfect forwarding을 위해서 필요하다. 자세한 내용은 추후 다룰 예정이다. 다음으로 rvalue, lvalue의 함수 오버로딩에 대해 알아보자

```
1 class X{};  
2  
3 // void foo(X x) { std::cout << "X" << std::endl }. // 값 타입과 참조 타입은 서로 오버로딩될 수 없다!  
4 void foo(X& x) { std::cout << "X&" << std::endl } // 1  
5 void foo(const X& x) { std::cout << "const X&" << std::endl }. // 2  
6 void foo(X&& x) { std::cout << "X&&" << std::endl } // 3  
7 //void foo(const X&& x) { std::cout << "const X&&" << std::endl }  
8  
9 int main(){  
10    X x;  
11    foo(x); // lvalue. 어느 곳에 오버로딩 해야할 지 몰라서 에러 발생  
12    // 값 타입을 주석처리하면 X x에 오버로딩 됨. (1, 2) 순서로 오버로딩  
13    foo(X()); // rvalue. 어느 곳에 오버로딩 해야할지 몰라서 에러 발생  
14    // 값 타입을 주석처리하면 X x에 오버로딩 됨. (3, 2) 순서로 오버로딩  
15 }
```

foo 함수의 마지막 표기법 const X&& x는 문법적으로는 가능하지만 사용하지 않는다. Move semantic을 통해 대체할 수 있기 때문이다. 다음은 주로 헷갈리는 문법에 대해 알아보자.

```
1 class X{};  
2  
3 void foo(X& x) { std::cout << "X&" << std::endl } // 1  
4 void foo(const X& x) { std::cout << "const X&" << std::endl }. // 2  
5 void foo(X&& x) { std::cout << "X&&" << std::endl } // 3  
6  
7 int main(){  
8    foo( X() ); // 3  
9  
10   X&& rx = X();  
11   foo(rx); // 3번이 호출될 것 같지만 1번이 호출됨! lvalue로 인식하기 때문  
12   foo(static_cast<X&&>(rx)); // 3  
13 }
```

X&& rx = X()에서 rx는 rvalue라고 생각할 수 있으나 **이름이 있으므로 lvalue로 취급된다.** 따라서 함수를 호출하면 1번이 호출된다!

따라서 3번을 호출하고 싶으면 static_cast<X&&>()을 사용하여 형변환을 하면 되는데 자세히 보면 **같은 타입을 왜 변환해야 하는가? 하는 의문이 생길 수 있다.** 이는 특수한 케이스로써 c++ 문법에서 타입 캐스팅이 아닌 value를 변환하는 캐스팅으로 기재되어 있다.

2.12.3 Reference collapsing

```
1 int main(){  
2    int n=3;  
3    int& lr = n; // lvalue reference  
4    int&& rr = 3; // rvalue reference  
5  
6    int& &ref2ref = lr; // error! 명시적으로 레퍼런스를 가리키는 레퍼런스는 코딩 불가  
7  
8    decltype(lr)& r1 = ? // int& & ==> int&  
9    decltype(lr)&& r2 = ? // int& && ==> int&  
10   decltype(rr)& r3 = ? // int&& & ==> int&  
11   decltype(rr)&& r4 = ? // int&& && ==> int&& 모두 두개씩 있을 때만 rvalue reference로 인식!  
12 }
```

레퍼런스를 가리키는 레퍼런스는 명시적으로 코딩이 불가능하다. 하지만 decltype()을 통해 타입을 추론하게 되면 가능하다.

`decltype(&&)&&` 인 경우에만 `int&&` 타입으로 추론하고 나머지 경우는 `int&`로 추론한다. 이런 추론 규칙을 **reference collapsing**이라고 한다! reference collapsing은 **typedef**, **using**, **decltype**, **template** 4가지 경우에 적용된다.

```
1 template<typename T> void foo(T&& arg) { }
2
3 int main(){
4     int n=3;
5
6     typedef int& lref;
7     lref&& r1 = n;           // int& && ==> int&
8
9     using rref = int&&;
10    rref&& r2 = 10;         // int&& && ==> int&&
11
12    decltype(r2)&& r3 = 10; // int&& && ==> int&&
13
14    foo<int&>(n);       // foo(int& && arg)
15    // foo(int& arg) 함수 생성!
16 }
```

2.12.4 Forwarding reference

```
1 void f1(int& arg) {}
2 void f2(int&& arg) {}
3 template<typename T> void f3(T& arg) {}
4
5 int main(){
6     int n=3;
7
8     f1(n); // ok
9     f1(0); // error
10
11    f2(n); // error
12    f2(0); // ok
13
14    f3(n); // ok. T는 어떤 값으로 받아도 lvalue reference이다.
15    f3(0); // error
16 }
```

`T&`는 어떤 타입으로 받아도 (`int&`, `int&&`) lvalue reference이다! 템플릿 케이스에 대해 보다 자세히 알아보자

```
1 template<typename T> void f3(T&& arg) \{\}
2
3 int main()\{
4     int n=3;
5
6     // 사용자가 명시적으로 타입을 추론한 경우
7     f3<int>(n);
8     f3<int&>(n);
9     f3<int&&>(n); // 3가지 케이스 모두 int&로 추론되므로 lvalue만 올 수 있다!
10
11    f3(n); // ok. 사용자가 템플릿 인자를 전달하지 않으면 int로 추론한다
12    f3(0); // error
13 }
```

다음으로 템플릿 인자에 **`T&&`**가 붙어있는 경우를 생각해보자.

```
1 template<typename T> void f4(T&& arg) {}
2
3 int main()\{
4     int n=3;
5 }
```

```

6     f4<int>(n);      // int &&. ==> int&& rvalue reference
7     f4<int&>(n);    // int& && ==> int&. 이 케이스에서만 lvalue reference가 된다!
8     f4<int&&>(n);   // int&& && ==> int&& rvalue reference
9
10    f4(n); // ok. 컴파일러가 자동으로 int 타입으로 변환해서 넘겨준다 (int ==> int)
11    f4(0); // ok.
12 }
```

T&& 와 같이 작성하면 타입 추론 규칙에 따라 rvalue와 lvalue를 같이 전달할 수 있다! 헷갈리기 쉬운 것이 함수가 하나인데 두 인자를 받는것이 아니라 함수가 2개가 생성되서 각각 따로 받는다. 그리고 생성된 각 함수는 call-by-value가 아닌 call-by-reference를 사용해서 전달받는다 이러한 T&& reference를 **forwarding(universal) reference**라고 부른다!

2.13 Move semantics

2.13.1 Move constructor

다음과 같이 얇은 복사(shallow copy)가 일어나는 경우를 살펴보자

```

1 class Person{
2     char* name;
3     int age;
4     public:
5     Person(const char*s, int a) : age(a) {
6         name = new char[strlen(s)+1];
7         strcpy_s(name, strlen(s)+1, s);
8     }
9     ~Person() { delete[] name; }
10 }
11
12 int main(){
13     Person p1("john", 20);
14     Person p2 = p1; // 얇은 복사 발생!
15 }
```

복사 생성자를 명시적으로 정해주지 않으면 컴파일러가 모든 멤버 변수 함수를 얇은복사하게 된다.

```

1 Person(const Person& p) : age(p.age) {
2     name = new char[strlen(p.name)+1];
3     strcpy_s(name, strlen(p.name)+1, p.name);
4 }
```

클래스 내에 다음과 같은 복사 생성자를 정의해주면 더 이상 얇은 복사가 발생하지 않고 **깊은 복사**가 발생한다! 하지만 복사 생성자는 성능 이유가 존재한다. 다음과 같이 임시 객체를 반환하는 함수를 살펴보자

```

1 class Person{
2     char* name;
3     int age;
4     public:
5     Person(const char*s, int a) : age(a) {
6         name = new char[strlen(s)+1];
7         strcpy_s(name, strlen(s)+1, s);
8     }
9     ~Person() { delete[] name; }
10    Person(const Person& p) : age(p.age) {
11        name = new char[strlen(p.name)+1];
12        strcpy_s(name, strlen(p.name)+1, p.name);
13    }
14 }
15
16 Person foo() {
17     Person p("john", 20);
18     return p;
19 }
20
21 int main(){
```

```
22     Person ret = foo(); // 임시객체 반환하여 ret에 복사 생성자를 호출하여 복사해주고 바로 파괴됨!
23 }
```

foo() 함수는 값을 반환하므로 임시객체가 생성되어 ret에 복사 생성자를 통해 깊은 복사를 일으키고 바로 파괴된다. foo()의 임시 객체를 파괴하지 않고 ret에 임시 객체의 주소를 그대로 가리킨 뒤 임시 객체의 메모리를 0으로 만드는 방법이 효율적이다!

```
1 Person(Person&& p) : name(p.name), age(p.age) {
2     p.name = nullptr;
3 }
```

Person 클래스 내부에 위와 같이 move 생성자를 만들면 rvalue만 받을 수 있고 임시 객체들은 해당 함수를 호출한다. 해당 코드가 rvalue를 처리하는 경우 기존의 복사 생성자보다 메모리 효율적이다!

2.13.2 std::move

```
1 class Object{
2     Object() = default;
3     Object(const Object& obj) { std::cout << "copy ctor" << std::endl; }
4     Object(Object&& obj) { std::cout << "move ctor" << std::endl; }
5 }
6
7 Object foo() {
8     Object obj;
9     return obj;
10 }
11
12 int main(){
13     Object obj1;
14     Object obj2 = obj1; // copy
15     Object obj3 = foo(); // move
16     Object obj4 = static_cast<Object&&>(obj1); // move
17     Object obj5 = std::move(obj2);           // move. 위 긴 변환문을 간단하게 move 함수를 호출하여 해결할 수
18     있다.
19 }
```

std::move 함수를 호출하면 obj1, obj2 같은 lvalue도 rvalue로 취급하여 move 생성자를 호출할 수 있다! obj1, obj2를 코드 내에서 더 이상 사용하지 않을 때 복사 생성자를 호출하기 보다 move 생성자를 호출하여 보다 효율적으로 값을 전달할 수 있다.

2.13.3 Move and noexcept

```
1 class Object {
2     public:
3     Object() = default;
4     Object(const Object&) { std::println("copy"); }
5     Object(Object&&) { std::println("move"); }
6 };
7
8 int main() {
9     std::vector<Object> v(3);
10    std::println("-----");
11    v.resize(5);           // copy, copy, copy 발생
12    std::println("-----");
13 }
```

v(3)로 처음 3개의 Obejct 자원을 확보한 후 resize를 통해 5개의 자원을 다시 확보한다고 5개 메모리를 새로 할당하고 기존 3개의 자원은 “복사(copy)”되어 새로운 메모리에 오게된다. 이렇게 vector의 베퍼를 새롭게 할당한 경우 결국 기존 베퍼를 제거하게 되므로 “복사(copy)”보다 “이동(move)”가 효율적이다! 하지만 위 코드를 실행하면 “복사(copy)”가 수행된다. 컴파일러가 기본적으로 복사를 수행하는 이유는 만약 이동을 하다가 예외가 발생하면 vector를 resize 이전 상태로 되돌릴 수 없다는 단점이 존재하기 때문이다.

따라서 이동을 사용하고 싶다면 되도록 예외가 발생하지 않도록 구현하고 noexcept 키워드를 붙여서 예외가 없음을

컴파일러에게 알려야 한다

```
1 class Object {
2     public:
3     Object() = default;
4     Object(const Object&) { std::println("copy"); }
5     Object(Object&&) noexcept { std::println("move"); } // 예외가 없음을 컴파일러에게 알림!
6 };
7
8 int main() {
9     Object o1;
10    Object o2 = o1;                                // copy
11    Object o3 = std::move(o1);                      // move
12    Object o4 = std::move_if_noexcept(o2);          // move
13
14    std::vector<Object> v(3);
15    std::println("-----");
16    v.resize(5);                                    // move, move, move 발생!
17    std::println("-----");
18 }
```

`std::move_if_noexcept`를 사용하면 컴파일러가 함수의 noexcept 키워드 유무를 검사하고 만약 키워드가 있다면 move를 실행한다. `std::move_if_noexcept`는 type_traits 기술로 예외 가능성을 조사한 후 예외 가능성이 있으면 const T& 타입으로 변환하고 예외 가능성이 없다면 T&& 타입으로 변환해주는 함수이다!

```
1 template<typename T>
2 constexpr std::conditional_t<
3     !std::is_nothrow_move_constructible_v<T> &&
4     std::is_copy_constructible_v<T>, const T&, T&&>
5 move_if_noexcept(T& x) noexcept {
6     return std::move(x)
7 }
```

다음과 같이 클래스가 여러 멤버 변수를 가지고 있는 경우를 살펴보자

```
1 template<typename T>
2 class Object{
3     int n;
4     std::string s;
5     T t;
6
7 public:
8     Object() = default;
9     Object(const Object& other) : n(other.n), s(other.s), t(other.t) {}
10    Object(Object&& other) noexcept
11        : n(other.n),
12        s(std::move(other.s)),
13        t(std::move(other.t))
14    {}
15 }
```

복사 생성자를 보면 `int n`은 예외가 없다. 그리고 `std::string s`은 예외가 없음을 보장한다. 하지만 **임의의 타입 T는 예외가 존재할 가능성이 존재한다.** 이럴 땐 다음과 같이 키워드를 입력하면 된다.

```
1 Object(Object&& other) noexcept( noexcept( t(std::move(other.t)) )
2     : n(other.n),
3     s(std::move(other.s)),
4     t(std::move(other.t))
5 )
```

c++에서 noexcept는 두 가지 의미가 존재한다. 1) noexcept operator, 2) noexcept specifier

1. noexcept operator:

- `bool b = noexcept(expression)`: 어떤 표현식이 예외 가능성이 있는지 조사한다

2. noexcept specifier:

- f() noexcept, f() noexcept(true): 함수 f는 예외가 없다.
- f() noexcept(false): 함수 f는 예외가 존재한다

또 다른 방법으로는 다음과 같이 쓸 수 있다.

```

1 Object(Object&& other) noexcept( std::is_nothrow_move_constructible_v<T> )
2   : n(other.n),
3   s(std::move(other.s)),
4   t(std::move(other.t))
5 {}
```

2.13.4 Default move constructor

move 생성자를 기본적으로 컴파일러가 제공하는 경우에 대해 살펴보자

```

1 class Object{
2   std::string name;
3 public:
4   Object() = default;
5   Object(const Object& other) : name(obj.name) {}           // [1] 복사 생성자
6   Object& operator=(const Object& obj) { name = obj.name; } // [2] 복사 대입 연산자
7   Object(Object&& obj) : name(std::move(obj.name)) {}       // [3] move 생성자
8   Object& operator=(Object&& obj) { name = std::move(obj.name); } // [4] move 대입 연산자
9};
```

1. **case 1.** 사용자가 [1,2,3,4] 모두 제공하지 않는 경우 컴파일러가 [1,2,3,4] 대한 default 버전을 제공한다.
2. **case 2.** 사용자가 [1] (또는 [2])만 제공하는 경우 컴파일러는 [2](또는 [1])는 default 버전을 제공하지만 [3,4]는 제공하지 않는다. 사실 [1]만 제공하면 컴파일러가 [2]도 제공하지 않는 것이 맞지만 설계 상 오류로 [2]는 default 버전이 제공된다. (since c++98)
3. **case 3.** 사용자가 [3](또는 [4])를 제공하는 경우 컴파일러는 [1], [2]를 삭제해버린다. 즉, 사용할 수 없다. 그리고 [4](또는 [3])는 제공하지 않는다.

하지만 코딩을 하다보면 복사 생성자는 사용자가 제공하지만 move 계열 함수만 컴파일리에게 요청하고 싶다.
이런 경우에는 어떻게 해야할까?

```

1 class Object{
2   std::string name;
3 public:
4   Object() = default;
5   Object(const Object& other) = default;      // 복사 대입 연산자도 move 생성자를 default로 요청하면
6   // 반드시 같이 요청해야 한다.
7   Object(Object&& obj) = default;
8   Object& operator=(Object&& obj) = default; // default 버전을 요청한다!
9};
```

(...)= default;로 default 버전을 요청하면 컴파일러가 move 생성자는 default 버전을 생성한다. 복사 대입 연산자도 move 생성자를 default로 요청할 때 같이 요청해야 한다! 위와 같은 코드 형태는 잘 작성된 오픈소스에서 많이 볼 수 있다!

2.13.5 Rule of 3/5/0

다음으로 널리 통용되는 규칙인 Rule of 3/5/0에 대해 알아보자

```

1 class Person {
2   char* name;
3   int age;
4 public:
5   Person(const char*s, int a) :age(a) {
6     name = new char[strlen(s)+1];
7     strcpy(name, strlen(s)+1, s);
8   }
9   // char*와 같이 포인터 멤버변수가 있도 동적으로 메모리를 할당한다면
```

```

10 // c++98 시절에는 소멸자/복사 생성자/복사 대입연산자를 반드시 만들어야 했다 (Rule of 3) !
11 // c++11 시절에는 소멸자/복사 생성자/복사 대입연산자/ move 생성자/move 대입연산자 또한 만들어야 한다.
12     (Rule of 5)

```

클래스 내부 변수에 포인터 멤버변수가 있고 동적으로 메모리를 할당한다면 반드시 선언해야 하는 함수를 가르켜 **Rule of 3 (c++98)**, **Rule of 5 (c++11)**라고 하였다.

char* 대신 std::string을 사용하면 사용자가 직접 자원을 관리할 필요가 없다. (= **Rule of 0**). 즉, STL에서 제공하는 클래스를 사용하면 클래스가 동적 메모리 할당에 대하여 컴파일러가 알아서 자동으로 제공한다. 결론은 STL을 많이 쓰면 좋다는 얘기이다.

2.14 Perfect forwarding

```

1 void foo(int n) {}
2 void goo(int& r) {r = 20; }
3
4 template<class F, class T>
5 void chronometry(F f, T arg) {
6     f(arg);
7 }
8
9 int main() {
10     int n = 10;
11     chronometry(foo, 10);
12     chronometry(goo, n); // error. T arg에 의해 값이 chronometry에 복사되어 goo에서 값을 20으로 바꿔도 n
13     값이 바뀌지 않는다.
14     std::cout << n << std::endl;
}

```

함수의 성능을 측정해주는 chronometry 템플릿 함수를 정의해보자. 이를 통해 두 함수 foo, goo를 하나의 함수 템플릿에서 처리하고 싶다. 이를 위해서는 perfect forwarding 기술이 필요하다. **Perfect forwarding**이란 전달 받은 인자를 다른 함수에게 “값, const 속성, value category 등의 변화없이 완벽하게 전달”하는 것을 말한다

rvalue 10과 lvalue n을 동시에 처리하기 위해 const T&를 사용할 수 있다. 하지만 이는 **전달 과정에서 const 속성이 추가되므로 완벽한 전달이라고 할 수 없다**.

```

1 void foo(const int n) {}
2 void goo(const int& r) { r=20; }
3
4 template<class F, class T>
5 void chronometry(F f, const T& arg) {
6     f(arg);
7 }

```

설명의 편의를 위해 우선 T를 integer로 고정해보자. lvalue와 rvalue를 속성 변화없이 그대로 받기 위해서는 다음과 같이 두 함수 템플릿이 필요하다.

```

1 template<class F, class T>
2 void chronometry(F f, int& arg) { // lvalue reference
3     f(arg);
4 }
5
6 template<class F, class T>
7 void chronometry(F f, int&& arg) { // rvalue reference
8     f(arg);
9 }

```

하지만 두 함수 템플릿은 hoo() 함수를 처리할 수 없다.

```

1 void hoo(int &&r) { }
2
3 template<class F, class T>
4 void chronometry(F f, int&& arg) { // rvalue reference
5     f(arg);
6 }
7

```

```

8 int main() {
9     hoo(10); // ok
10    chronometry(hoo, 10); // error
11 }

```

chronometry에서 rvalue reference int&&로 받아도 arg라는 이름이 생기기 때문에 더 이상 rvalue가 아니게 된다. 즉, 10은 rvalue이지만 arg는 lvalue이다.

```

1 template<class F, class T>
2 void chronometry(F f, int&& arg) { // rvalue reference
3     // f(arg);
4     f(static_cast<int&&>(arg)); // 다시 rvalue로 캐스팅해서 보내야 함!
5 }

```

따라서 rvalue를 호출할 때는 static_cast로 다시 한 번 rvalue로 캐스팅해서 넘겨줘야 한다.

2.14.1 Using forwarding reference

Forwarding reference를 사용하면 int&, int&&를 자동 생성할 수 있다. 따라서 함수 템플릿을 두 개 생성하지 않아도 된다. 하지만 템플릿을 사용하려면 구현부가 동일해야 하는데 하나는 캐스팅이 있고 하나는 캐스팅이 없다.

```

1 void chronometry(F f, int& arg) { // lvalue reference
2     f(static_cast<int&>(arg));
3 }
4
5 template<class F, class T>
6 void chronometry(F f, int&& arg) { // rvalue reference
7     f(static_cast<int&&>(arg));
8 }

```

따라서 lvalue에도 캐스팅을 추가해준다. 컴파일 타임에서 lvalue 캐스팅은 같은 타입 캐스팅이기 때문에 무시된다. 이제 구현부까지 동일해졌으므로 하나의 템플릿 함수로 합칠 수 있다.

```

1 template<class F, class T>
2 void chronometry(F f, T&& arg) { // rvalue reference
3     f(static_cast<T&&>(arg));
4 }
5
6 int main() {
7     chronometry(goo, n); // T = int&, T&& = int&
8     chronometry(hoo, 10); // T = int, T&& = int&&
9 }

```

Perfect forwarding은 const까지 자동으로 커버해주기 때문에 const 전용 함수를 만들지 않아도 된다. C++ 표준에는 캐스팅을 대신 해주는 함수가 존재한다. static_cast를 사용하는 대신 `std::forward` 함수를 사용한다.

```

1 template<class F, class T>
2 void chronometry(F f, T&& arg) { // rvalue reference
3     f( std::forward<T>(arg) );
4 }

```

`std::forward` 함수는 forward referencing을 자동으로 해주는 함수이며 lvalue를 (함수로 전달하면) lvalue로 캐스팅하고 rvalue를 (함수로 전달하면 받으면서 lvalue로 변경된 것을 다시) rvalue로 캐스팅해준다. 즉, **인자가 rvalue 일 때만 std::move()를 하는 것을 의미한다!**

2.14.2 Variadic parameter template

만약 foo와 goo의 파라미터 개수가 다르다면 어떻게 해야 할까? template에서 가변인자를 받는 ...을 사용한다.

```

1 void foo() {}
2 int& goo(int a, int& b, int&& c) {
3     b=20;
4     return b;
5 }

```

```
6
7 template<class F, class ... T>
8 void chronometry(F f, T&& ... arg) { // rvalue reference
9     f( std::forward<T>(arg) ... );
10 }
```

goo는 리턴값이 존재한다. 이런 경우 어떻게 해야할까? auto를 사용하게 되면 참조값을 버리는 특성이 있으므로 **decltype(auto)**를 사용한다.

```
1 template<class F, class ... T>
2 decltype(auto) chronometry(F f, T&& ... arg) { // rvalue reference
3     return f( std::forward<T>(arg) ... );
4 }
```

Perfect forwarding을 정리하면 다음과 같다

1. 인자를 받을 때 **forwarding reference (T&&)**를 사용한다.
2. 인자를 다른 함수에 전달할 때 **std::forward<T>(arg)**로 묶어서 전달한다.
3. 여러 개의 인자를 모두 forwaring하기 위해 **가변인자 템플릿**을 사용한다.
4. 반환 값도 전달하기 위해서는 **decltype(auto)**로 반환한다.

3 References

- [1] (lecture) CODENURI - C++ Master
- [2] (blog) [모던C++] 정리 - tango1202

4 Revision log

- 1st: 2024-07-16
- 2nd: 2024-07-23
- 3rd: 2024-07-28